AD NUMBER

AD923470

LIMITATION CHANGES

TO:

Approved for public release; distribution is unlimited.

FROM:

Distribution authorized to U.S. Gov't. agencies only; Test and Evaluation; 02 JUL 1974. Other requests shall be referred to Ballistic Missile Defense Advance Technology Center, Attn: ATC-P, P.O. Box 1500, Huntsville AL 35807.

AUTHORITY

USABMDSC per ltr, 3 Dec 1975

22944-6921-020

SOFTWARE REQUIREMENTS ENGINEERING METHODOLOGY NOTEBOOK

CDRL G00C

OCTOBER 31, 1974

(FINAL REPORT FOR DAHC60-71-C-0049)

Sponsored By
BALLISTIC MISSILE DEFENSE
ADVANCED TECHNOLOGY CENTER
DAHC60-71-C-0049

TRW
SYSTEMS GROUP
Huntsville, Alabama

22944-6921-02.

SOFTWARE REQUIREMENTS ENGINEERING METHODOLOGY NOTEBOOK

CDRL GOOC

OCTOBER 31, 1974

(FINAL REPORT FOR DAHC60-71-C-0049)

Sponsored By
Ballistic Missile Defense
Advanced Technology Center
DAHC60-71-C-0049

TRW
SYSTEMS GROUP
Huntsville, Alabama

SOFTWARE REQUIREMENTS ENGINEERING METHODOLOGY NOTEBOOK

OCTOBER 31, 1974

CDRL GOOC

(FINAL REPORT FOR DAHC60-71-C-0049)

Approved By:

T. W. Kampe

T. W. Kampe, Manager
Software Requirements
Engineering Program

James E. Long

James E. Long, Manager
Huntsville Army Support Facility

Sponsored By
Ballistic Missile Defense
Advanced Technology Center
DAHC60-71-C-0049

TRW
SYSTEMS GROUP
Huntsville, Alabama

TABLE OF CONTENTS

INTRODUCTION

The Ballistic Missile Defense Advanced Technology Center (BMDATC) has established the Software Development and Evaluation Program in order to develop a complete and unified engineering approach to software development and testing ranging from synthesis of system specifications through completion and testing of the process design. A key element of this program is the Software Requirements Engineering Program (SREP)--a research program concerned with the development of a systematic approach to the development of complete and validated software requirements specifications. Primary objectives of this research are to ensure a well-defined technique for the decomposition of system requirements into structured software requirements, provide requirements development visibility, maintain requirements development independent of machine implementation, allow for easy response to system requirements changes, and provide for testable and easily validated software requirements.

To meet these objectives, the Software Requirements Engineering Methodology is being developed. This methodology is an integrated, structured approach to the requirements engineering activities from the parsing and translation into data processing software requirements of system requirements provided in a System Requirements and Performance Specification (SRPS) through analysis, refinement, validation and documentation of the software requirements in a Process Performance Requirements (PPR) Specification.

The purpose of this notebook is to collect and present in a single document significant research results from the methodology development activities. The notebook form will facilitate the planned addition of future results from the methodology research, thereby providing an evolving information base on the methodology and forming the basis for a final, complete methodology description to be completed under subsequent contracts. Contained herein are five separate documents previously delivered to BMDATC as drafts for review and approval prior to inclusion in this notebook. Each document appears as a separate chapter, and is included as a stand-alone report as approved by BMDATC.

Chapter 1, "Special Report - Current Software Requirements Engineering Technology", contains the results of a review and evaluation of the current technology for the development of large scale software, with special emphasis on requirements engineering. This technology evaluation provides a necessary basis for the research activities by 1) showing that no existing methodology satisfies the SREP requirements and 2) providing a data base on concepts, techniques and software tools that may be applicable.

An overall description of the planned research is contained in Chapter 2, "Software Performance Requirements - Software Requirements Engineering Methodology". This chapter presents a basic problem statement and description of the anticipated research activities and products. Because research directions and objectives can change as intermediate results are obtained and assessed, the plans presented in this section may require revision in the future.

Chapter 3, "Software Capability Description - Software Requirements Engineering Methodology", documents the capabilities required in the methodology and forms the basis for a more detailed design of the methodology. The problem description from Chapter 2 is further analyzed and expanded; key assumptions concerning the types of systems to which the methodology is to be applied are discussed; and fundamental concepts which form the basis for design of the methodology are presented.

Chapter 4, "Test and Evaluation Procedure - Software Requirements Engineering Methodology", identifies the tests and experiments to be performed using the methodology and the procedures to be used to evaluate their success. The rationale for the design of the tests and experiments is provided; the interactions and objectives of each experiment and test are summarized, procedures for carrying out the experiments and documenting the results are summarized; and descriptions of the planned experiments and tests provided.

Basic requirements on the Requirements Statement Language (RSL) are contained in Chapter 5, "Software Performance Requirements - Software Requirements Language". In addition to the methodology for requirements development and specification, an integrated software system, Requirements Engineering and Validation System (REVS), will be developed to support the approach. In conjunction with and parallel to the methodology and REVS capability definition,

a Requirements Statement Language (RSL) will be defined consisting of the syntax and semantics necessary to define software requirements and to control REVS. Chapter 5 defines the fundamental concepts, general requirements and desired characteristics of that language. The approach to be used in the language development, general requirements on the language, specific requirements on semantics, and techniques for language implementation are addressed.

22944-6921-010

CURRENT SOFTWARE REQUIREMENTS ENGINEERING TECHNOLOGY

CDRL G00A

AUGUST 15, 1974

Sponsored By
BALLISTIC MISSILE DEFENSE
ADVANCED TECHNOLOGY CENTER

DAHC60-71-C-0049

TRW
SYSTEMS GROUP
Huntsville, Alabama

22944-6921-010

CURRENT SOFTWARE REQUIREMENTS

ENGINEERING TECHNOLOGY

CDRL G00A

AUGUST 15, 1974

Sponsored By
Ballistic Missile Defense
Advanced Technology Center

DAHC60-71-C-0049

TRW
SYSTEMS GROUP
Huntsville, Alabama

CURRENT SOFTWARE REQUIREMENTS ENGINEERING TECHNOLOGY

CDRL G00A

AUGUST 15, 1974

Principal Author

F. Burns

Principal Contributors

T. Bell
F. Herring
M. Zazulak

Approved By:

T. W. Kampe

T. W. Kampe, Manager
Software Requirements
Engineering Program

James E. Long

James E. Long, Manager
Huntsville Army Support Facility

Sponsored By
Ballistic Missile Defense
Advanced Technology Center
DAHC60-71-C-0049

TRW SYSTEMS GROUP
Huntsville, Alabama

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

LIST OF ILLUSTRATIONS

LIST OF TABLES

1.0 INTRODUCTION

1.1 OBJECTIVES

The report is designed to evaluate the existing research and development that is ongoing in large scale software programs. It covers several specific areas of research that are particularly applicable to the BMDATC research program. Effort was directed at the following areas:

- System development methodologies,
- Languages,
- Testing techniques,
- Management techniques, and
- Automated tools.

Each of these five areas was examined with special emphasis on requirements engineering and the early stages of software development. We found that the concept of automated tools permeated much of the literature in the other four areas. Therefore, the report integrates material about automated tools into the sections on methodologies, languages, testing, and management techniques.

In the following, methodology is intended to be the sum of four parts:

1) A collection of basic ideas indicating a certain spirit or viewpoint,
2) Techniques which apply the basic ideas,
3) A set of tools to use in employing the techniques to "real-world" problems, and
4) An overall procedure describing when to use the tools and techniques.

This definition is intended to distinguish between the words methodology, techniques, and tools to indicate the degree of completeness of the items examined. When it is convenient to be less specific, we shall generically refer to these items as disciplines, procedures, methods or systems.

The purpose of this evaluation was to determine whether a methodology exists that is applicable to the development of large scale real-time software requirements. Disciplines which may have appropriate features and applicable techniques are considered. Languages are examined with a view toward requirement statements and simulation construction. Software testing methods have been explored to determine if any methods are applicable to requirements testing, especially those being used to validate and verify large programs. Various techniques to manage large software development programs have also been examined since these techniques may have even a larger effect on program outcome than technical considerations.

The sources for this survey include the traditional ones:

- Software trade publications,
- Technical research, survey publications and newsletters,
- Proceedings of major, special, and minor software conferences,
- Important books,
- Contract reports, and
- Consultation with experts in the fields.

The procedures and methods of the government agencies, Army, Air Force, Navy, NASA, FAA, etc. have been assimilated indirectly through the sources above.

Various judgements, rankings, and ratings have been made on several of the items discussed herein. These decisions have been based only on the available information which seldom included examples of realistic applications. In fact, one might suggest at this point that an indicator of how well developed is a methodology or portion thereof is the amount of information available. However, we believe that we collected enough information on these items so that the relative comparisons are essentially correct.

1.2 OVERVIEW AND CONCLUSIONS

It can be stated without any qualifications that there is no methodology implemented or any collection of techniques and tools which is applicable to the entire problem of real-time software requirements engineering and the early stages of software development. Several issues lead one to this conclusion. There is no methodology defined in sufficient detail that tackles the problem of real-time software requirements and software development. Those methodologies that are applicable (even though partially) are not completely implemented. Scientific applications are deemphasized and intensive real-time computational needs are generally unmentioned. Little regard is given to the issue of computer independence and, in fact, most of the systems examined introduce computer-dependent elements early in the requirement analysis process. Systems generally have little emphasis on how to state requirements. Those examined are either a concise description of a methodology without any apparent implementation or a specific implementation of some general ideas never formally stated. Summarizing, it seems that only recently it has been realized that a great many software errors are directly attributed to invalid software requirements and a poor software requirement development procedure. However, more attention in the software development world is being directed at software requirements analysis and there is currently an active pursuit of many different segments of this research area.

ISDOS [M.24]* and SODA [M.22] represent a broadside attack on the system definition problem by means of problem statement languages and analyzers. The simulator construction system, AUTASIM [M.6], provides a contribution to the feasibility question of automating the process of building simulators. The eclectic Model-Driven approach [M.4] represents a crystallizing of a total approach to the software development process. Many automated tools are available but are not integrated into a usable system.

*The alphanumeric characters in brackets refer to the list of references in Section 6.0.

The management techniques provided by the methodologies are little more than nil at best. One reason for the paucity of any management analysis in this area is the lack of meaningful data on large software projects. Comparison of large software projects with large business production-oriented projects is not possible as pointed out by Weinwurm [M.32]. The well-developed management techniques for production oriented environments are just not applicable to research and development atmospheres (as large software projects seem to be). The Chief Programmer Team concept of Baker [M.2] has been applied very successfully to medium software projects. The integration of Mill's [T.22] Top Down structured programming with Baker's Chief Programmer Team into their Team-of-Teams approach [M.1] to address large-scale software projects shows some promise, however, there is no published data currently to indicate success or failure.

The testing activity is a necessary part of a software requirements methodology. However, the survey indicates that the testing of software requirements is relegated solely to the regime of software testing. The Problem Statement Language and Problem Statement Analyzer of the ISDOS project seems to be the only directly applicable activity found. Even though the methods are fairly elementary (e.g., I/O checking, documentation, etc.) ISDOS represents a beginning in requirements testing. Hetzel [T.12] points out that software testing, debugging, and reliability assessment is an embryonic activity with regard to any whole or unified theory. One can only conclude from this that testing software requirements is at best in the same town as software but it may be even across the tracks. However, validation and verification experts are usually very optimistic people, for the following reasons:

- There are portions of software testing that are well developed (see e.g. [T.12] and [T.26]) and are making cost-effective contributions to software development regularly.
- All areas of software testing, debugging, and reliability assessment are undergoing vigorous research activities (see [T.23] and [T.7]).
- Software requirements testing seems to be only an isomorphic mapping away from software testing.
- Simulation provides a medium for using software to validate software requirements.

All of these reasons provide motivation and confidence in a testing activity. It is commonly acknowledged that testing should be integrated very early in a requirements methodology; both in the action of the methodology as it applies to software development and in the action of providing the methodology with the testing features.

In the immediate future, it would seem most profitable for requirements testing of a methodology to proceed in the following directions:

- Determine a well-defined structure for software requirements.
- In tandem with the above, establish a machine readable specification language in which requirements may be unambiguously stated.
- Determine a medium (i.e., a realistic model of the software development process) in which the automated testing tools may be integrated.

With a view toward supporting a methodology with a high level language, a comprehensive list of language requirements has been compiled (see Table 3.1) and used to judge a selected number of representative candidates. Two major themes dominate this list. One, it is desired that the language communicate software requirements succinctly and second, necessary simulation elements must be specified. A methodology and its supporting language are bound so tightly that one would expect that no single language would fulfill all of the requirements in this list unless it was connected with some methodology. The survey bears this fact out. Each requirement is well satisfied by some particular language, but no single language satisfied all. The most readily applicable language that has been examined is the simulator construction language AUTASCRIPT of the AUTASIM system. If not directly usable as a language, the techniques and procedures of AUTASCRIPT are certainly usable.

Most of the higher level languages have been plagued with a set of problems that are definitely related to the two themes mentioned above. These can be stated as: 1) the development of a syntax that is easily readable for the non-professional programmer and at the same time, is adequate for such technical usage as the expression of algorithms; 2) the selection of a character set sufficiently large to support such a syntax and yet not defy implemenation; and 3) implementation that generates efficient code

within a reasonable compile time. Attempts to resolve these difficulties, in particular the latter, range from the implementation of language subsets only to the use of two compilers, one for checkout and one for optimizing. The results of all these efforts has been a reduction of language power. The most promising area to find a language in support of a software requirements methodology would appear to be that of extensible languages. These allow the user to expand the language to suit his particular needs without the overhead of a general purpose language.

2.0 PROCEDURES FOR SYSTEM ANALYSIS

This section discusses and evaluates various techniques, tools, and methodologies used today for software requirements analysis and for the early stages of software development.

2.1 OVERVIEW AND COMMENTS

The fact that research efforts are concentrating more on system analysis and less on automated programming is highlighted in a survey article by Head [M.12]. Head claims that simulator languages and systems are the most notable achievements in system analysis today. Even though there has been little day-to-day impact on the system analyst, he predicts that major changes will evolve from this research activity. Head points out that the early attempts in system development techniuqes (e.g., decision tables, SOP) have been generally unaccepted and the more contemporary methods (e.g., ARDI, TAG, Formatted File Organization) have been useful but not sufficiently encompassing. High regard is given to ISDOS for its far reaching goals and final judgement must wait.

An historical perspective of system development techniques is offered by Couger [M.9]. The computer generations are used to time-line the spectrum of system methods. Couger points out that the evolution of system analysis techniques has lagged computer hardware evolution by one full generation. He addresses the periods from the first generation through the third, and for the fourth generation the spirit and substance of ISDOS permeate Couger's predictions. He concludes by claiming the gap between automated system methods and development of hardware will be substantially narrowed by the advent of the fourth generation of computers.

Evidently, many researchers are impressed with the scope of the ISDOS project. A segment of the ISDOS activity was directed at a review of the literature on systems building, Teichroew and Carlson [M.27]. They point out that most publications in this area start without referring to other references and develop their own terminology and description of the process. Only occasionally are there comparisons with other publications. They conclude that a system development cycle is a fact of life in most medium and

large scale organizations. In spite of the apparent differences, the processes as currently practiced have many common features and Teichroew and Carlson devise a model of the system building process incorporating the observed commonalities.

One might observe, here, that Teichroew and Carlson are victims of their own criticism by introducing a new system building model, however, their actions have the virtue of integrating a collection of seemingly different processes.

Our intention, here, is to interpret the remarks above in view of applicability to software requirements development.

2.2 COMPARISON OF SELECTED METHODS

Most of the system building techniques mentioned by Head [M.12], Couger [M.9], and Teichroew, Carlson [M.27] were examined along with other techniques that were felt to be directly applicable software requirements development. These items were examined and reviewed in detail before the final list that is discussed in this section was selected. The fourteen that were finally chosen cover several different characteristics of the software development process. Several were chosen because they were the most widely used at the present time. Among these were the Accurately Defined System (ADS) by NCR, the Time Automated Grid system (TAG) by IBM, and the Hierarchy and Input Process Output system (HIPO) by IBM. Top-Down programming was chosen because it is the most publicized structured approach in the literature.

Others were chosen because they represent significant advancements of the state-of-the-art. Among these are the Information Systems Design and Optimization System (ISDOS), that is under development at the University of Michigan; LOGOS, which has been under development at Case Western Reserve; AUTASIM, which was developed by General Research Corporation for the Army Logistics Command; and the Systems Optimization and Design Algorithm (SODA) which was initiated at Case Western Reserve and is currently under development at Purdue. Each of these methodologies reflects a unique and valuable contribution to the development of requirements and/or software.

Some were included even though the discipline is not specifically oriented toward software development because they have techniques, or checklists, or compilations of techniques which can be beneficial or useful during some phases of software development. An example of this type of methodology is ARDI which stands for Analysis, Requirements determination, Design and development, Implementation and evaluation. ARDI was developed as a management systems handbook by a number of people working for Phillips Gloeilampenfalorieken.

Not all of the systems are directly related to the development of software. LOGOS and AUTASIM are directed toward design of a computer software system and the automated construction of a simulator, respectively. However, both have features which are of direct interest to the BMDATC research program.

Two disciplines, CSC Threads and Chief Programmer team of IBM, were selected because they deal directly with management techniques. The Model Driven approach was included since it represented an encompassing methodology dealing with the entire software development process. IBM's Study Organization Plan was included because it was an early attempt to formalize the steps of the system development process. TRW's Engagement Logic approach deals specifically with the software requirements regime and provides useful groundwork.

A more detailed description of each of these is contained in Section 3.3.

The basic method of conducting the comparison is to examine the disciplines in five areas: product orientation, scope in system development cycle, principle concepts, degree of automation, and supporting languages. Each of these issues is addressed separately in the following.

2.2.1 Product Orientation

Each of the fourteen procedures was first examined to determine what type of information system they could be used most effectively to develop (see Figure 2-1) and the magnitude of information system they could be used on. In every category, each discipline was scored according to their capabilities. The scoring given (here, and in Figures 2-2 thru 2-5) was an "H" if the item was highly appropriate, an "M" if it had some applicability but wasn't specifically oriented toward the category, an "L" if it had a low applicability, and, if no capabilities were directly applicable to the category, the cell was left blank.

System Development Type

Information systems were subdivided into six major categories: simulation, real-time software, management information systems, commercial systems for business applications, data reduction systems, and telecommunications systems.

The two most appropriate for simulation were AUTASIM and LOGOS. AUTASIM was developed specifically to automate the construction of a simulation for the Army Logistics Command. It features a language called AUTASCRIPT that enables the analyst or simulation developer to communicate

			TAG	ADS	ISDOS	SODA	LOGOS	HIPO	CSC THREADS	TOP DOWN	MODEL DRIVEN	CHIEF PROGRAMMER	SCP	ENGAGEMENT LOGIC	ARDI	AUTASIM
PRODUCT ORIENTATION	SYSTEM TYPE	SIMULATION		L			H	H	L	M	M	M			M	H
		REALTIME SOFTWARE	L		L			L		M	M	M		H	L	
		MANAGEMENT INFO	H	H	H			H	H	M	L	H				M
		BUSINESS APPLICATION	H	H	H	H		H	H	M	M	H	H		H	
		DATA REDUCTION	L	L				M	H	M	M	M			M	
		TELECOMMUNICATION	M	M	M			L	M	M	M	M			L	H
	SYSTEM SIZE	LARGE PROG. >100K	L		M			L	L	H	H	M	L	H	L	
		MEDIUM 10 TO 100K	H	H	H	H	H	H	H	H	H	H	H	H	H	M
		SMALL <10K	H	H	H	H	H	H	H	H	H		H	H	H	H

Figure 2-1 Product Orientation

directly with the machine and automatically link modules to be included in the simulation. It is the only example found of completely automatic simulation construction. AUTASIM's application is limited; large systems such as encountered in Ballistic Missile Defense may be difficult to simulate in the same manner because of the complexity of the control networks involved, or because the language that is required to specify the simulator for a BMD system is extensive and complex requiring many verbs in the syntax. LOGOS is basically designed to simulate computer systems and to aid in the conceptual and detailed design of a computer system. It features a number of computerized design tools that are already in existence. Another system that appears to be directly appropriate to the development of simulation systems is HIPO. It is basically a diagrammatical approach to developing a system design and is applicable to all six categories of product orientation. The Top-Down approach to software development of course applies to simulation as does the Chief Programmer concept. ARDI also has some checklists that are useful aids in determining whether the necessary information for building the simulation has been accumulated. The others have low applicability to simulation.

None of the items directly address the problem of developing large scale real-time programs. The Engagement Logic and Model Driven approaches, are designed to provide requirements for real-time software. The Chief Programmer team concept has been demonstrated on a number of programs and can be applicable to any of the information systems being discussed here. TAG, ISDOS, and HIPO all have some general applicability to real-time software development. The principal problem with each is that they fail to provide a mechanism for specifying the timing requirements that are inherent in the process. Something else that should be noted; in the Top-Down, the Model Driven, or the Chief Programmer approach the separation of personnel talents from the methodology itself is often indistinguishable. Thus, the success of the methodology is often as dependent upon the individuals that are using the methodology as upon the methodology itself.

TAG, ADS, ISDOS, SODA, HIPO, CSC Threads, SOP, and ARDI are all oriented toward management information systems or business applications. All of these feature disciplined approaches to developing the requirements in software design for an information processing system. TAG is used to

determine input and output and to create the data for the automated program. The Accurately Defined System also utilizes specific forms that are filled out by the analyst to develop the flow of information through a system. HIPO uses a set of diagrams in a specified manner to portray the system design and flow of information through a system. The same thing is true of SOP and ARDI. ISDOS is one of the most advanced methods for developing information processing systems. It features a requirement statement language, and problem statement analyzer (PSA) which allows the system designer to specify his requirements and then have those requirements analyzed by the PSA. A feature of ISDOS that should be noted is that all of the analysis is performed independent of any computer implementation. Ultimately, SODA will be included as a part of ISDOS. SODA is basically a methodology for automating a systems design function in the development of an information processing system. It features procedures to select the required resources as well as alternative designs of program structure and file structure. Procedures for selecting auxiliary memory devices and optimization and performance evaluation schemes are also a feature of SODA. SODA is also oriented toward commercial information systems.

It should be noted that there is a basic difference between commercial information systems and real-time processes. Both can generally be described in terms of input, processing, and output. The commercial information systems emphasize the input and output with large data bases. The real-time systems such as BMD and the air traffic control problem, feature primarily large processing schemes to command the external environment. The processing is triggered by stimuli and the output is a command to a system external to the data processing system. This difference in orientation was emphasized in our review of systems developed for commercial information systems, such as TAG and HIPO, where the forms for the diagrams are designed to emphasize the data bases and the input and output, rather than a high degree of processing as exists in a command and control system. This different orientation means that a system oriented toward commercial applications is more difficult to adapt to a real-time software project. However, many of the techniques are applicable and of good use to the development of real-time software systems. To highlight this difference, we might note that according to IBM, "TAG is a general purpose technique applicable to the design of any data processing system in the commercial environment" [M.15].

Data reduction systems such as those designed to reduce data from manned spaceflights or test vehicles, or for large scale evaluation of specific weapon systems can and do have special techniques and tools designed for them. One of these is CSC Threads which features an automatic reporting system that enables program management to keep abreast of the program development and control the system configuration during the software development cycle for large programs. TAG, ADS, HIPO, and ARDI all have low applicability to data reduction systems, however, by their very nature of introducing discipline into the software design and development they can be useful tools. Similarly, the Top-Down, Model Driven, and Chief Programmer approaches can be used in developing data reduction systems with beneficial results in terms of increasing productivity and reducing development cycle time.

Telecommunications is another system category that these disciplines can be oriented toward. AUTASIM specifically was designed to simulate telecommunications systems. Many of the others, as can be seen in Figure 2-1, have some applicability though none were specifically derived to automate or aid the design of a telecomminications systems.

System Development Size

The next set of attributes that was considered in determining the product orientation of each of these methods was the size of program the systems were intended to develop. An arbitrary set of classifications was developed; a small system was considered to be less than 10,000 instructions; a medium system was considered to be greater than 10,000 but less than 100,000 instructions. Almost all the disciplines were designed to handle medium to small systems. The only one that is questionable about handling a medium-sized system is AUTASIM where certainly the system being simulated can be quite large in terms of data base but perhaps not as large in terms of instructions. The ongoing research, however, is oriented toward large programs, those greater than 100,000 instructions. Site Defense is said to have over 140,000 instructions, the Safeguard program and air traffic control systems would be of the same magnitude. Except possibly for ISDOS, most of the other systems have very little applicability to large programs.

2.2.2 Scope in System Development Cycle

Understanding the system development process can be facilitated by distinguishing the major steps. From beginning to end we are concerned with the following five phases: system concept, software requirements analysis, software design, software implementation, testing, and operations. The steps in the development cycle covered by the selected items is illustrated in Figure 2-2. Most of these methods are concerned with the systems concept, requirements analysis and software design. In fact, only the Chief Programmer, the Model Driven approach, and the Top-Down approach are really concerned with software implementation, the testing and the operation of the software itself. The ADS system and HIPO are primarily oriented toward systems concepts and the computer-independent requirements analysis and contribute little to the software design, and almost none to the software implementation, testing or operations. AUTASIM, TAG and ISDOS have little contribution to the development of a systems concept. However, TAG and ISDOS both are highly oriented towards requirements analysis and the drawing of an initial system concept down into more specific detail. From the standpoint of the currently available research data, the majority of the systems studied certainly contribute to the analysis of requirements for an information system. SODA and LOGOS, however, are primarily oriented toward a computer-dependent software design. The Engagement Logic approach was specifically developed to bridge the gap between a top-level system design and a computer-dependent software design.

2.2.3 Principle Concepts

Development of each of the fourteen systems was guided by certain underlying concepts and ideas. These are summarized in Figure 2-3. We used the system of marking those items "H" that had a high orientation toward the principle concept, "M" reflecting less emphasis, and a blank to represent no known emphasis.

Eight concepts were identified, ranging from the desire to improve the capability for systems design to a desire to describe the inputs, processes, and outputs. All of the disciplines, naturally, ranked at least M in the desire to improve the capability for systems design.

STAGE OF INFO. SYS. DEV.		TAG	ADS	ISDOS	SODA	LOGOS	HIPO	CSC THREADS	TOP DOWN	MODEL DRIVEN	CHIEF PROGRAMMER	SOP	ENGAGEMENT LOGIC	ARDI	AUTASIM
	SYSTEM CONCEPT	L	H	L		H	H	M	H			H	H	M	L
	SOFTWARE REQT'S — COMP. IND.	H	H	H			H	H	H			H	H	M	M
	SOFTWARE REQT'S — COMP. DEP.	H			H	H	H	H	H			H	L	M	M
	SOFTWARE DESIGN	H		H	H		H	M	H	H	H	H	H	H	H
	SOFTWARE IMPLEMENTATION								H		H				
	TESTING								M	M	M				
	OPERATIONS								M	M	M				

Figure 2-2 Stage of System Development

	TAG	ADS	ISDOS	SODA	LOGOS	HIPO	CSC THREADS	TOP DOWN	MODEL DRIVEN	CHIEF PROGRAMMER	SOP	ENGAGEMENT LOGIC	ARDI	AUTASIM
IMPROVE CAPABILITY FOR SYSTEM DESIGN	M	M	H	H	H	H	M	M	H	M	H	H	H	M
SIMULATE SYSTEM DESIGN AUTOMATICALLY					H									
AUTOMATE SIMULATOR CONSTRUCTION														H
IMPROVE COMMUNICATION OF REQT'S	H	H	H			H			M		M	H	M	
SYSTEMATIZE SYSTEM DESIGN		H				H					M		M	
IMPROVE REQUIREMENTS ANALYSIS	M		H			M			H		H		H	
INTRODUCE DISCIPLINE	M	M				H					M		M	
PROVIDE MODULAR BUILDING BLOCK						M								H
DESCRIBE INPUTS, PROCESSES, OUTPUTS	H	H	L			H	H						M	

Figure 2-3 Principle Concepts

In the event a particular system placed emphasis on improvement of requirements analysis, it usually derived directly from the desire to improve the capability for system design. On the other hand, an original concept of improving a systems design does not automatically produce improved requirements analysis. For instance, SODA, LOGOS, and AUTASIM all have as an underlying motivation the improvement of the systems design through automated analysis techniques or through simulations. SODA, for instance, tries alternative designs to an information system to select the best system design. LOGOS develops a candidate system, both hardware and software, and analyzes the candidate system capabilities. Thus, several system designs can be tried and evaluated prior to actually preparing the specifications for the real information systems. AUTASIM also builds a simulator based on some candidate system design and evaluates the results for that simulation. Neither SODA, LOGOS, or AUTASIM has as a basis for development improved requirements analysis. However, all of them help evaluate the system design and can be used to develop a set of requirements based on the design having the most potential effectiveness.

LOGOS and AUTASIM were built on the concept of simulation of systems design. In LOGOS due to a wider scope, simulator construction is only semi-automatic while AUTASIM being more narrow in scope provides complete automation. Both of these systems have met with some success in these concepts.

TAG and ADS were designed to improve the communication of requirements. A common problem is that systems designers and those responsible for the programs do not speak the same languages and use different terminologies, consequently there is an inability to communicate the true requirements from the system designer to the programmer. Both TAG and ADS are specialized forms that are completed by the system designers which determine the input and the output of the information system being developed. In the case of TAG, and ADS when it is automated, the forms can be translated into machine input and analyzed for completeness. They begin with the output of information processing system and work backwards through the system to determine the necessary inputs and the processes that will be required to get from the input to the output. Both TAG and ADS have as a goal the description of the input, processes, and outputs of the target information system and

they serve this concept well. They also introduce some discipline into the process of developing an information system.

SOP was originally designed to improve the capability for describing and developing a systems design and to improve, in part, the communication of requirements and systemize the system design. It does serve to improve requirements analysis because it requires a complete history and overview of any existing information systems and of the organization that the information system will be designed to serve. It also introduces some discipline, though to a lesser degree than many of the other methodologies. It too uses forms that must be filled out by the analyst while developing the candidate system design. It was not originally intended to be automated and has not subsequently been subjected to any automation.

ARDI was originally designed to provide a handbook for developing management information systems. Thus it can be said that it provides an improved capability for systems design. It does this through checklists and forms and by describing various techniques that can be used to develop a systems design. ARDI also provides an improved requirements analysis through the same set of capabilities. In addition, it does show how the inputs, processes, and outputs can be described and how discipline can be introduced into the information systems development.

ISDOS provides for the improved communication of requirements. It does so by having a requirements language which is analyzed by a Problem Statement Analyzer, subsequently producing requirements in a form and format that are easy to understand and which represent the true requirements of the system in greater detail. This is done through the Problem Statement Analyzer and the other tools that are being developed, which include a systems optimizer and design analyzer capability. The total set of tools is designed to analyze data requirements between elements of the proposed information system, identify discontinuities, and to perform a static analysis of the requirements to ensure that they are a complete set of requirements. This static analysis also determines whether the data from one element is the proper input to another element.

Many of the methods examined showed facility for describing inputs, processing, and outputs in great detail. Engagement Logic, HIPO, CSC Threads,

TAG, ADS, and SODA all provide for describing data and control flow and primary paths. Provision for overall modularity was a major emphasis of AUTASIM, LOGOS, Top-Down programming and to a lesser degree HIPO.

2.2.4 Degree of Automation

A desirable characteristic to ask of any formalized procedure is to provide automation wherever appropriate. The benefits are obvious (e.g., elimination of errors, shortened response times, allowance for more creative thinking, etc.) consequently this aspect is considered. The conclusions of these considerations are shown in Figure 2-4.

A significant point to remember when reviewing the chart (Figure 2-4) reflecting the automation inherent in the systems is that some of them were not designed with automation in mind. HIPO possesses no automation, neither does the Study Organization Plan. And while the Top Down, the Model Driven Approach, and the Chief Programmer team concept show no indication of automation, they are techniques which are applicable to large scale software programs and which have features that would lend themselves to automation. Also many of the things that contribute to a successful software development program are manual systems such as the unit development folder in use on the Site Defense program and the Chief Programmer team that put the New York Times system on line [M.1].

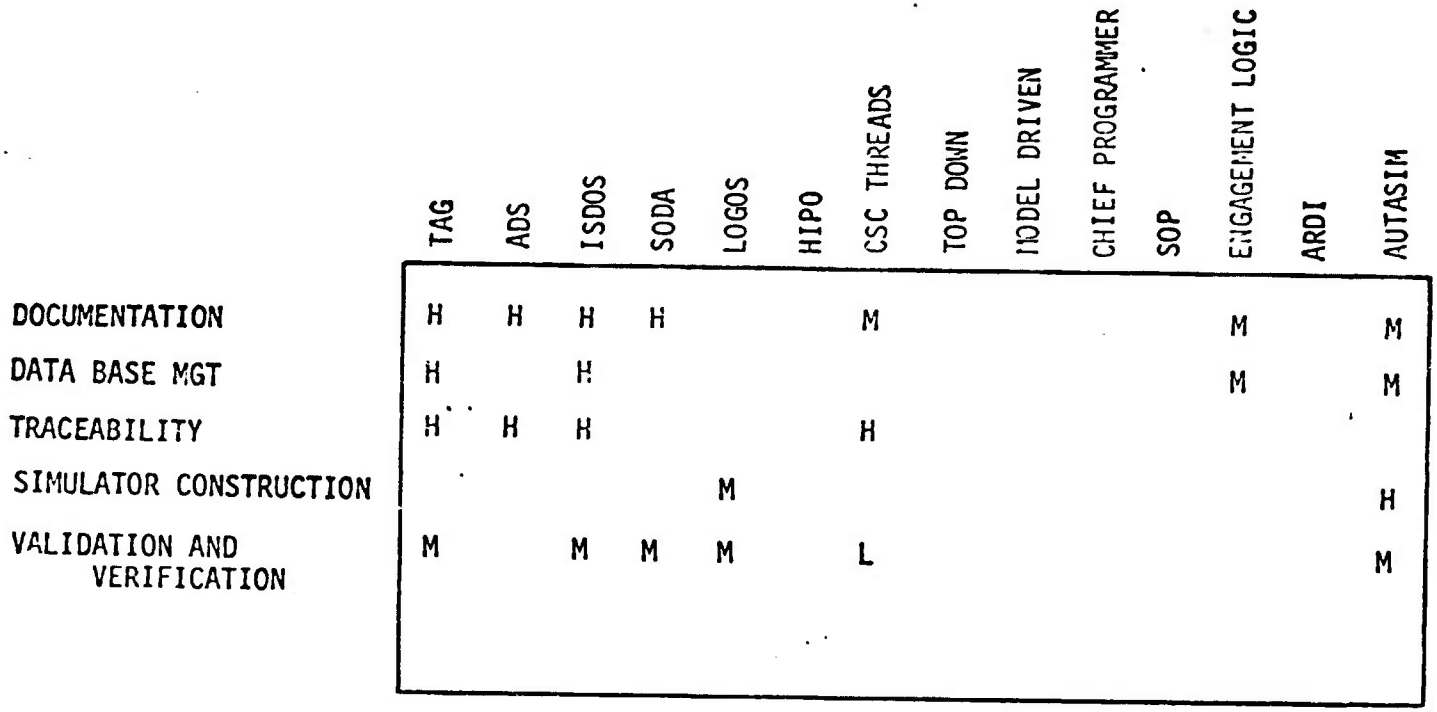
Although automated tools and techniques

- will introduce a discipline into the requirements and software development and
- will vastly improve certain aspects of the information system development, and reduce programming and man hours in many phases of the information system development

there still

- will be some manual techniques that are appropriate for the software development.

TAG, ADS, ISDOS, and SODA have excellent documentation features that display the requirements in a format that can be readily understood by the analyst and software developers. The format and contents of the output can be selected by the analyst prior to any computer runs. To a lesser degree, CSC Threads, Engagement Logic, and AUTASIM also have the capability to



	TAG	ADS	ISDOS	SODA	LOGOS	HIPO	CSC THREADS	TOP DOWN	MODEL DRIVEN	CHIEF PROGRAMMER	SOP	ENGAGEMENT LOGIC	ARDI	AUTASIM
DOCUMENTATION	H	H	H	H			M					M		M
DATA BASE MGT	H		H									M		M
TRACEABILITY	H	H	H				H							
SIMULATOR CONSTRUCTION					M									H
VALIDATION AND VERIFICATION	M		M	M	M		L							M

Figure 2-4 Degree of Automation

document analysis. TAG provides documentation of the file formats and system flow plus output to show the analyst the criteria for optimizing the data flow. It produces as an automatic by-product standardized, up-to-date documentation. CSC Threads produces documentation of the requirements and management information in various formats that can be selected by the project management. It also prohibits unauthorized access to the management information.

Because of their emphasis on the input and output data of the information systems, both TAG and ISDOS provide many automated capabilities to analyze data base requirements. For instance, TAG permits the user to determine missing data, redundant data, the minimum data base, and any discrepancies in the use of data. In addition, file formats and information needed to optimize the data flow are provided. TAG also permits the analysis of volume specifications and the computer hardware requirements that result from these volume specifications, plus it provides a dynamic analysis to indicate the time dependent relationships of the data.

ISDOS provides many of the above features plus a data and function dictionary that can be used in the logical systems design. These features are automatically output from the Problem Statement Analyzer. Tools that were developed with Engagement Logic have capabilities to analyze the data base that the system the Engagement Logic represents would require. The data is initially input in a specified format for each function. The inputs and outputs are examined to ensure that there is a proper data flow between functions and inconsistencies are output for review by the data base design.

AUTASIM and the Engagement Logic are the only two systems that have the capabilities to automatically generate data bases. AUTASIM generates the data base with the language and the modules that have been created prior to the simulator construction. The Engagement Logic data base still begins with manual generation but is automatically incorporated into the simulator design through the use of a special purpose program. In addition, an automated tool is available to show the data interfaces between functions in the Engagement Logis so that inconsistencies in data requirements can be detected.

Traceability is an all important feature if management is to know that the requirements that he started with have been incorporated into the final software produce or into intermediate levels of the software development cycle. Automated traceability reduces the probability of human error and increases the possibility of getting a true mapping of requirements throughout each level of the software development stage.

Four of the systems exhibit high degrees of automation in providing traceability between the requirements and the software that is developed. These were TAG, ADS, ISDOS, and CSC Threads. Each of these provide traceability from one level to the next and to some degree provide traceability at each level back to the original system objectives.

CSC Threads was originally designed to provide management visibility through traceability. It is highly automated and performs reporting capabilities that can inform management of the status of individual functions that are to be developed or the total system. It also provides configuration control mechanisms on the software being developed, so that software can be entered into the system with a set of key words so that access by unauthorized individuals can be denied, ensuring the status quo of the software configuration.

The validation of requirements through simulation is a necessary part of any methodology that will be used to develop large, complex real-time systems. The capability of automating all or portions of the simulation or of activities that are generally associated with simulations was considered a feature that should be reviewed.

AUTASIM proved to be the only system found that had the capability to automatically generate simulations. The scope and type of simulations that can be developed with AUTASIM are limited to logistics systems in a telecommunications environment. While the applicability to a large scale real-time system is not direct, the techniques and philosophy that were used in creating AUTASIM certainly warrant further attentions and study. The basic concept underlying AUTASIM is that of modules, which represent the function that must be performed by the simulation of a logistics system, being predeveloped and resident internal to the AUTASIM system. These modules, then, are subject to call when the appropriate calling sequence is created

by the AUTASCRIPT language. The language serves to link the modules together and create part of the data base for the simulation.

Although not completely automated, LOGOS provides some simulator construction aids by allowing macro insertions in its supporting language.

Automated tools are supplied by some systems for assisting in validating and verifying requirements beyond simulation. The Problem Statement Analyzer of ISDOS that checks for connectivity between modules. It also checks to see if data output from one module matches the input to a successor module to determine whether there are incompatabilities. The primary function is to provide rudimentary facilities for computing volume specifications and the time dependent relationships of the data in the problem statement. TAG also has mechanisms to examine the input/output data sets to identify redundant and missing data in the information flow.

The tracing of requirements on one level of the system development cycle to the next succeeding level is a verifying activity. Only CSC Threads and TAG appear to possess any capability for automatically performing this type of verification of requirements. CSC Threads is designed to keep track of requirements during all phases of an information systems development. There are special computer programs in the CSC Threads system that perform this function. The data is initially entered through manual efforts on prescribed forms that are converted to computer input that is fed into these programs.

2.2.5 Supporting Languages

The degree of which a partial collection of techniques and tools or a complete methodology is well integrated is indicated by the scope of their supporting languages. Various language attributes of the fourteen disciplines is discussed below. The remarks are partitioned into two parts: first, the language use in simulation and software and secondly, the language use in requirements analysis. The essential comments are summarized in Figure 2-5. The major language features are selected from those compiled in Section 3. An immediate conclusion one reaches is that only three of the systems examined psssessed a supporting language. SODA may ultimately have the language of ISDOS due to the fact they are scheduled to be merged at a later date.

			TAG	ADS	ISDOS	SODA	LOGOS	HIPO	CSC THREADS	TOP DOWN	MODEL DRIVEN	CHIEF PROGRAMMER	SOP	ENGAGEMENT LOGIC	ARDI	AUTASIM
LANGUAGE	SIMULATION AND SOFTWARE	DATA RECORDING														H
		REPORT GENERATION			H		M									H
		SIMULATOR EXECUTION					H									H
		SIMULATOR CONSOLIDATION					H									H
		FUNCTIONAL MODELING			M		M									H
		READIBILITY/USEABILITY			H		L									M
		DATA BASE MGT.			L		M									H
		LIBRARY MGT.														H
	REQUIREMENTS	REQUIREMENTS SPEC.			H		M									
		TESTING					H									
		DOCUMENTATION GEN.			H											
		MGT. VISIBILITY														
		NATURALNESS			M											M
		STRUCTURED			L		H									L

Figure 2-5 Supporting Languages

AUTASIM, ISDOS, and LOGOS offer features supporting use in simulation and software. AUTASIM and LOGOS both have automatic data recording devices that are used to analyze simulations. That this capability exists in the language is not clear, though apparently the output can be selected in both. The problem statement language of ISDOS does provide a significant report generation capability. AUTASIM also provides a high degree of report generation capabilities, while LOGOS provides somewhat less. Each is oriented toward specific types of output that is selectable through the syntax of the language. Various formats are available and can be selected using the languages of each methodology. Since LOGOS and AUTASIM are designed to build simulations they both have numerous language features which are intended to aid the execution of a simulator or the consolidation of simulator elements, results, and/or performance data. The linkage features and delimiters that are available in AUTASCRIPT, the language of AUTASIM, have specific application to the development of simulators using modules that can be called in to the simulation from internal storage. LOGOS has the capability to include macros, structures such as IFTHENELSE, and capabilities to define the system through a graphic console. The language for AUTASIM, AUTASCRIPT, is a much higher order language than that of LOGOS. The LOGOS language can be compared to ALGOL 68. Because of their language features both LOGOS and AUTASIM provide the capability to functionally model performance of specific functions that would be a part of the system being designed. In the case of LOGOS, it is through macros, and in the case of AUTASIM, it is through modules that can be linked together using the language. ISDOS also allows some functional modeling and yet this is not oriented toward the building of simulations or of software.

In requirements analysis ISDOS exceeds the others. The ISDOS language is designed specifically to analyze requirements and to specify these requirements in terms of statements that can be automatically analyzed by the Problem Statement Analyzer. AUTASIM is used to build simulators that analyze requirements so it can be said to have an application to requirements specifications though that was not the original intent of the language. For the same reason, LOGOS has features that can be used to analyze requirements through simulations. ISDOS has commands that enable specification documents to be generated. Thus, it also possesses a high ranking in the documentation of systems specifications.

Two important features that should be considered when discussing any language that is going to be used to specify requirements are their naturalness and their structure. ISDOS and AUTASIM both possess a degree of naturalness. The ISDOS language features a syntax close to that of the English language. It must be used with certain constraints on the order of the words, the number of words that are in the vocabulary, and the manner in which it may be used. The flexibility of the AUTASIM language can be increased with additional syntax. However, at present it is limited to logistics application and there are a number of constraints on how the syntax can be used to develop a simulation. The LOGOS language is the most structured with the ISDOS Problem Statement Language and AUTASIM less structured.

2.3 DESCRIPTIONS OF SYSTEM ANALYSIS METHODS

A summary of each item considered in the review is presented. This will provide the reader with a more complete description of each discipline. In addition, references to more detailed descriptions for each are given.

2.3.1 TAG - (Time Automated Grid System)

TAG [M.15] was developed by IBM originally as a manual system design tool and was automated late in 1966. Tag according to IBM is, "a general-purpose technique applicable to the design of any data processing system in the commercial environment". Its intent is to reduce the time required to design a system through systematizing the study effort and by providing forms and automated techniques.

The rationale for developing TAG is

> "The key to effective design lies in giving over more of the available study time to the decision-making processes and making it easier for the systems planner to reach conclusions about data requirements. If the systems analyst must rely on his own data recording and organizing abilities (as he had to do in the past), his productivity cannot be as great in the purely inventive area of design. What is needed is a technique to relieve him of his clerical duties while directing him in his creative efforts. That technique is TAG."

TAG basically works from the output of a system, and given those, works backwards through the system to determine the required inputs. With TAG the user can determine data flow, redundant data, minimum data base and discrepancies in the use of any data. Iteration of the program can produce file formats and system flow descriptions based on the time when data enters the system and subsequently is required to produce output. The user is provided information which enables him to add needed outputs and/or optimize data flow.

"TAG is an iterative tool; its function is to develop an integrated systems flow and to maintain that integration no matter how many changes or how much additional data the user introduces."

"To summarize, use of TAG permits a reduction in the time and effort required to go from problem definition to systems implementation, maximizes the creative use of systems personnel, and produces as an automatic by-product standardized, up-to-date documentation."

The principal limitations to TAG are that it was designed for business applications, emphasis is on the input and output, and timing and accuracy requirements are difficult to determine and specify.

2.3.2 ADS - (Accurately Defined Systems)

ADS [M.17] was developed by NCR starting in 1966. Initially, the intent was to apply a uniform discipline to the communication links between the system specification and implementation so that the industry specialists at NCR could convey information system needs of industry to the programmers who were to develop these packages. This need for clear communication arose because of the introduction of a new computer system with the attendant requirement to supply user-oriented software packages.

NCR feels that ADS as it evolved, has become a general purpose tool. "That is, although the technique was developed to solve a problem in connection with a specific line of computers, this phase of the effort is completely hardware independent."

"ADS (Accurately Defined Systems) is a technique for communication and establishment of a working discipline in the definition of objectives, criteria, and specifications for systems being designed for EDP processing. The ADS method is not oriented to a particular brand of hardware. It approaches system definition by starting with specification of the report form or document which is to be output. With the report as a starting point, separate, interrelated forms are used to

specify input records, history records, computation, and logic operations. All system elements are tied together through continuing use of cross reference procedures. The result is a disciplined, universally understandable set of documents suitable for use in programming systems on any computer hardware. Communication between programmers and users of computers is standardized and disciplined for universal validity and understanding".

ADS is oriented toward business uses of electronic data processing and commences after the feasibility studies have been performed. It deals with the system documentation only. The Xerox Corporation has a project underway to automate the ADS system.

2.3.3 ISDOS - (Information System Design and Optimization System)

ISDOS ([M.27], [M.28], [M.29]) is currently being developed by the Department of Industrial and Operations Engineering at the University of Michigan. Faculty, students and research associates under the leadership of Professor D. Teichroew are participating in the development.

"ISDOS begins with the user requirements recorded in a machine readable form. The problem definer (i.e., the analyst or the user) expresses the requirements according to a structured format called the Problem Statement Language." The Problem Statement Language (PSL) is designed to provide understandable communication and documentation between man and machine through a simple syntax. The language is designed so that the user can express what characteristics he desires of the software system input, output, and processing. Restrictions are composed to prohibit the user from doing anything past a logical systems design.

The PSL is designed to be operated on by a Problem Statement Analyzer (PSA) which checks for the correct syntax and then produces a data dictionary and a function dictionary that are useful in the logical system design.

The PSA also "performs static network analysis to ensure the completeness of the derived relationships, dynamic analysis to indicate the time-dependent relationships of the data, and an analysis of the volume specifications". The analyses are done independent of any computer implementation of the software.

Incorporated into the PSA are operations research methods that make it possible to evaluate candidate designs or design strategies and also provide an indication of how the system can be expected to perform. There is also a Data Re-Organizer, a Code Generator, and a Systems Director in ISDOS. The Data Re-Organizer stores data in the specified formats and on the specified devices. The Code Generator organizes the problem statements into programs, reorganizing the data interfaces that were specified by the Data Re-Organizer, and then generates either machine code or higher level language statements. The Systems Director takes the output of the other modules and proceduces a target information processing system.

The principal concept upon which ISDOS is being built is the separation of user requirements from decisions on how the requirements should be implemented. Principal interest has been on the information processing systems to serve management and hence, the application to real-time systems would require changes to ISDOS.

2.3.4 SODA - (Systems Optimization and Design Algorithm)

SODA [M.20] was developed at Case Western Reserve University. It is basically a discipline for automating the system design functions of the development of an Information Processing System (IPS). SODA starts with a statement of the processing requirements and has as its objective the automated generation of a complete systems design. SODA, given a number of submodels optimizes the total design by using mathematical programming, graph theory, and heuristic procedures. The algorithm is a multi-level decision model with the decision variable at one level becoming a constraint at the next level.

It is assumed the problem definer can identify his requirements which he then states in a problem statement language. The requirements are then analyzed and organized by a Problem Statement Analyzer. Next alternative system designs are generated using SODA/ALT, which is a procedure for

the selection of a CPU and core size as well as alternative designs of program and file structure. Following this, SODA/OPT is exercised. SODA/OPT is a procedure for the selection of auxiliary memory devices and the optimization and performance evaluation of alternative designs.

SODA outputs a list specifying which of the available computer resources will be used, specifications of the programs generated, specifications of the file structure and the devices on which they will be stored, and a schedule of the sequences in which the programs must be run to accomplish the stated objectives.

SODA/PSA follows the papers of Borje Langefors ([M.17], [M.18]) which deal with the use of matrix algebra and graph theory to represent the processing and data units in an IPS and Raymond Briggs [M.5] which provides the necessary structure to develop a program and File Structure Algorithm.

SODA has been partially incorporated into the ISDOS system being developed by Teichroew at the University of Michigan. The programs are on the Univac 1108 at Case Western Reserve, the IBM 360/67 at the University of Michigan and the CDC 6500 at Purdue University.

2.3.5 LOGOS

LOGOS [M.23] is under development at Case Western Reserve University where the principal investigator is Professor E. L. Glaser. It is basically a computer-aided design (CAD) system for the conceptual and detailed design of computer systems. Implementation has been on a Digital Equipment PDP-10 with a Bolt, Beranek and Newman paging box and TENEX executive system. Work on LOGOS was sponsored in part by the Advanced Research Projects Agency of the Department of Defense.

LOGOS has as goals "the creation of a multi-designer environment in which computer system designers can define a system in which a high degree of parallelism or concurrency exists, verify its logical and functional consistency, evaluate its expected performance before implementation, and finally implement the hardware and generate the code for the software". To achieve these goals LOGOS provides a representation system that has a well defined syntax and schematics and permits the target system structure to view an ordered hierarchy of layers.

The highest layer is the interface with the system users and the lowest may be machine language instructions or library subroutines or, on a hardware system, NAND gates.

The representation scheme in LOGOS is sufficiently general to allow both hardware and software facilities to be represented by algorithms. The representation is specific enough to allow algorithm consistency and performance analysis and is decomposable into elements which may be implemented directly. Since many of today's computer systems contain parallel processing capabilities, the representation allows specification of parallelism or concurrency.

LOGOS uses a graph-theoretic system of representation that synthesizes and extends the works of Petri [M.21], Karp and Miller [M.16], and others. A cursory treatment of the representation can be found in [M.20] and more computer treatment in [M.22] and [M.23].

LOGOS allows the system designer to define the system capabilities in the LOGOS design data base through graphics consoles. These capabilities may include MACROS, structures such as IFTHENELSE, or MSI functions for hardware. Initially this may be at the system level but as the description becomes computed, the design is carried to the next layer and defined in more detail. Each designer's work is made available to the other designers through a common global data base. Modifications can also be evaluated by substituting the mods into the data base. The process is continued until the functions or capabilities are reduced to primitives (e.g., machine language instructions). Hardware and software are developed across the same layers and actual building of the hardware and software occurs after tradeoffs and performance analysis of the target system is conducted at the levels where system primitives are identified. In addition, LOGOS provides the designer with several types of consistency and performance analyses.

LOGOS has a highly-structured, non-natural language similar to ALGOL 68 which may impose too restrictive a set of rules on the requirements analyst. Ambiguities are basically prohibited.

2.3.6 HIPO - (Hierarchy and Inputs, Process, Output)

HIPO [M.13] is an IBM-developed procedure for graphically developing, and thus describing, the design and implementation of software programs. It is designed to alleviate the problem of narrative documentation with resultant misinterpretations, and to produce an ordered set of design documents which can be readily accessed. Since it is a highly visual system, everyone associated with the development of a program or programming system can be informed of the design and program requirements.

HIPO describes the design in terms of inputs, processes, and outputs of functions. A visual table of contents of all documentation and overview diagrams of the functions are first developed. As detailed diagrams at each hierarchical level are developed, the table of contents and overview diagrams are iterated upon to maintain accurate pointers from the requirements stated on the overview to all lower level designs. An adjunct to each detailed diagram is a description, in narrative form, of the steps in the process. This provides additional information on the processing required. There are symbols with specific meanings for data transfer, control flow, access data, detail picture, etc.

Emphasis in HIPO is towards the input and output rather than the process. Consequently, it is more suited toward the design and implementation of programs with large data base manipulation requirements. Timing and accuracy requirements and the statement of analytical processing steps could not be easily handled with the HIPO diagrams.

"Hierarchy" is included in the name because HIPO develops the design in a top-down approach. Emphasis is placed on having consistent detail, terminology, and graphic techniques at each level. The inputs, processes, and outputs are developed manually in simple diagrams. The diagrams get more detailed as design progresses but they remain fairly simple throughout the design process.

There are three kinds of HIPO packages. One is for the initial design package, the next is for the detailed design package, and the final is for program documentation.

2.3.7 CSC Threads

CSC Threads [M.7] was developed by the Computer Sciences Corporation to increase the effectiveness of development management information systems. It is based on the concept that functional requirements trace a path through the system. The system is broken into subsystems of the systems elements at the top level. These subsystems may be software, hardware, manual operations or combinations thereof and are partially dependent on the size and complexity of the system being developed.

A functional requirement is identified and traced through the subsystems or system elements at the top level to produce top-level threads which have a definable stimulus and response. At the next level, the same process is followed except the functional requirement is traced through the information on the various levels of threads just discussed that are useful and necessary to manage the development of a software system. The information is traceable to the top level threads. In addition, there are features that permit tracking of groups of threads which may be an increment in the development of the total system.

The use of "Threads" requires that a specific technique be used in developing a new system thereby forcing the systems engineer into a disciplined approached to defining the requirements on the system. Forms and worksheets to aid the threading of functional requirements through the system are provided as are forms for developing subsystem and task level threads. These forms have places to fill in the appropriate management information and provide an intermediate step for entering the information into the set of programs which are a part of "Threads".

Status reports are output from the programs to aid project management, configuration management, quality assurance, and the implementers.

A syntax has been developed for interfacing the system designers with "Threads". Security measures are incorporated into the programs to restrict the access to the management information. The programs are implemented in CSC's BASIC language on the INFONET system. "Threads" basically provides management information and is oriented toward non-real-time systems.

2.3.8 Top-Down Programming

Top-down programming is a programming method proposed by H. Mills [T.21]. The technique even though primarily programming oriented has definite management procedure implications. Along with Baker, Mills [M.1] has attempted to integrate top-down programming, structured programming of Dijkstra, and Baker's Chief Programmer teams into a cohesive highly specialized "team of teams" approach to managing large software projects.

The top-down approach requires that programming proceed from developing the control interface statements and initial data definitions downward to developing and integrating the functional units.

2.3.9 Model Driven Software

Model driven software [M.4] is a methodology in which "activities are guided by an explicit, consistent model of how software should be organized, developed, and maintained." There are several examples of different software approaches which illustrate this idea. DOD's baseline-milestone configuration management guidelines address development and documentation aspects. Top-down, structured programming provides a model for the software design and development process as well as for the control structure of the resulting software. ISDOS, TAG, and other business systems models address user and data organization issues.

Model driven software is based on four complementary submodels.

1) A model of the software development and maintenance process,

2) A model of characteristics of quality software,

3) A model of information processing performance, and

4) A structural model of the software, including both program control and data aspects.

None of the examples mentioned in the first paragraph encompass four such models. However, the examples do have valuable models and guidelines to offer. The model driven approach is a conceptual integration of the best parts of the available techniques.

2.3.10 Chief Programmer Teams

Chief Programmer Teams [M.2] is a management organization of production programming projects as proposed by Baker. Chief Programmer Teams represent a change in management approach from a loosely structured group of programmers to a highly structured team of programming specialists who work under strict operational discipline. The overall didcipline is composed of three parts.

- A standard team nucleus consisting of a chief programmer, back-up programmer, and a programming secretary.
- A technical discipline based on structured programming.
- An organizational discipline based on a central visible programming support library.

IBM has used this management procedure in programming for the New York Times information bank. Results of this experiment are very encouraging. Baker points out that some questions are still open. Will the procedure work for larger (than The New York Times information bank) systems? For most jobs, can sufficient experience and competence be found to fill the senior level, chief programmer positions? Will chief programmers be willing to accept the technical and managerial challenges of large projects with few people?

2.3.11 SOP - (Study Organization Plan)

The Study Organization Plan [M.10], [M.14] was developed by IBM and is based partially upon a set of reports which can be used to document the three phases of the study and design of a system. These phases are "1) understanding the existing system, 2) determining the true requirements of the system, and 3) designing and describing a new system to fulfill the true requirements".

Data is collected in major categories such as: history and framework, industry background, goals and objectives, policies and practices, and government regulation.

The types of data and information sources are described along with the value and purpose for gathering the data. Rules for filling out each form in the system are provided along with a documentation structure for developing information on the processes which must be performed by the system. The forms are designed so they may be used at different levels of detail, with different input, or in different phases of the study and design of a system. The use of a standard set of documents also "substantially eases the problem of communicating the h- -teristics of a system or installation." The forms are designed to guide the developer in a logical review of the current system and introduces a degree of discipline into the development of the true requirements. The activity requirements are developed using a model showing the required relationships within the activity. An activity is made up of operations. An operation is defined as something which, when initiated by a trigger, converts inputs to outputs, and uses resources to effect this transformation.

SOP is oriented to developing business information systems and does not have any automated features. It does introduce discipline and standards to study and develop the system requirements.

2.3.12 EL - (Engagement Logic)

Engagement Logic (EL) ([M.30], [M.31]) was developed by TRW Systems as a method of specifying software requirements for the Hardsite Ballistic Missile Defense System. EL specifies how the system responds to a Ballistic Missile threat during an engagement. The following are identified:

- The various decisions which must be made,
- The mathematical procedures required, and
- The data needed to make the decisions and to perform the mathematical calculations.

Engagement Logic is directed at identification of operational requirements which must be satisfied by the software. In developing EL, an identification of all functions which the system must perform is made and a clear picture of the functions required to process a class of objects entering the system is developed. These are shown in flow diagrams which display the processing flow and the data required at each processing step.

The flow diagrams (called functional sequence diagrams, FSD's) contain the functions, decision points which separate or link the functions, and the interactions between functions. Associated with the FSD's are narratives which delineate the processing required in each function, subfunction, or decision point. Contingency and recovery paths are also identified. EL constitutes a conceptual design of the process which the software must implement. As such it provides a vehicle for requirements integration and interface control and provides a basis for system trade studies. EL can be functionally simulated to study the system response to the postulated threat scenarios.

A set of automated tools has been developed to aid in the development of EL. Among these are a data base generator and documentation aids. Specific symbols and formats are used to facilitate readability and understandability. In addition, there are development tools which are appropriate for BMD systems and which if modified may have applicability to more general systems. A candidate design is developed very early in the system design and subsequently developed in increasing detail until a process design for the software is developed.

2.3.13 ARDI - (Analysis, Requirements Determination, Design and Development, Implementation and Evaluation)

ARDI [M.11] is a systems engineering approach to developing management information systems which was developed over a decade of computer experience at Phillips Gloeilampenfalorieken in Eindhoven.

The handbook describing ARDI covers the gamut for developing software from conception to installation and operation, including physical constraints for the equipment and personnel. Series of checklists, forms, diagrams, and procedures are given to aid the system designer develop not only the requirements, but also the programs, hardware requirements, the physical facility, etc. A general outline for applying these features is also provided. In addition, alternatives are presented in many instances so that there is a great deal of flexibility available. The choice of alternatives is left to the system developers.

The ARDI approach is aimed toward business applications. It takes advantage of the existing system to identify functions that are essential to the realization of system objectives. These functions are synthesized at the top-level and broken down into subsystems. This top-down approach continues until the subsystem can be developed independently.

Three candidate methods for developing information flow are given. These are the departmental method which breaks the organization down into the existing departments, the stimulus response method which starts with a trigger (stimulus) and follows the information flow through until response, and finally the action oriented which is the study of the information required to initiate or control specifications.

2.3.14 AUTASIM - (AUTomated Assembly of SIMulation Models)

AUTASIM [M.6] was developed by the General Research Corporation (GRC) as a component of the MAWLOGS system (Model of the Army Worldwide Logistic System). Dr. Robert T. Burger says, "that the system is designed to rapidly assemble discrete event, stochastic simulation models of node network system". Main components are a library that contains modules, or models, of logistic system components, a program to assemble the modules, and a model description language called AUTASCRIPT.

AUTASIM is based on the GASP II (General Activity Simulation Program) structure. Modifications to make an unconventional use of assigned GO TO statements to permit recursive reentry to FORTRAN subroutines were made so that the full flexibility of the AUTASIM linkage system could be utilized. One feature of AUTASIM which is notable is the use of a "modular, building block concept that enables models to be assembled for a very wide range of systems". Thus, the system can be adapted to other uses provided the objective (i.e., a discrete event simulation) is the same and appropriate models are constructed and entered into the library. Statistical data collection capability is automatically included if desired by the analyst. Reports from a generalized output capability can also be selected at execution time and is subsequently produced by the system.

In addition, a capability to run the model from a cold start is included. The capability properly initializes model conditions before statistics collection is begun.

The model description language "is a special purpose code which is accepted by the Model Assembler Program and is used to define the content and structure desired in the simulation model". The code has rules for fitting modules together. Each module has a name that is part of the vocabulary of the code. Delimiters are provided so that a very complex multi-function node network can be described.

AUTASIM represents one of the few automated simulator constructor programs in the literature and thus deserves inclusion in the analysis of current methods. Many of the features of AUTASIM appear highly desirable in any methodology for producing software. However, the limitations or constraints of the system should be recognized. Among these are: the fact that each model has to be developed and placed in the library individually (this probably is true for any automated simulator constructor), there is no capability to apply time constraints to the simulator, and the simulator construction is basically implemented through a process construction language.

3.0 LANGUAGE SURVEY

The following questions are addressed in this section: What properties of a computer language are necessary to adequately support a requirements methodology? Is there a language with these properties and if not, can one be designed and constructed?

3.1 OBJECTIVES AND COMMENTS

In order to support a requirements methodology reasonably well, a computer language must be able to

- communicate a system's requirements succinctly and
- specify procedural elements for simulation.

Clearly, these two objectives overlap, but there are areas which are not common to both (see Figure 3-1). For example, in stating requirements it may be necessary to specify documentation standards which has no impact in simulation. On the other hand, the specification of a simulator requires computer dependent statements as well as design decisions beyond those contained in a list of requirements. But this aspect presents no real dilemma to a language since those elements not common to both areas are simply dealt with in a different manner.

Two survey articles on simulation languages were of particular interest. DesRoches [L.18] has provided a compendium of languages elaborating on each, its major features, applications, and references. No comparisons, rankings, or judgements are given but several references are supplied which deal with a few languages in a comparative manner. Kiviat [L.34] discusses four languages in some detail: GPSS, SIMSCRIPT II, SIMULA, and CSL. These languages were felt to be representative of the different aspects of simulation techniques. Kiviat indicates that the research activity is generally partitioned into two areas: the development of new simulation concepts and the development of improved simulation systems. Future research areas in simulation languages were suggested. The two areas most prominent were (i) the unifying of discrete-event and continuous-time simulation languages and (ii) new ways of system modeling both static and dynamic.

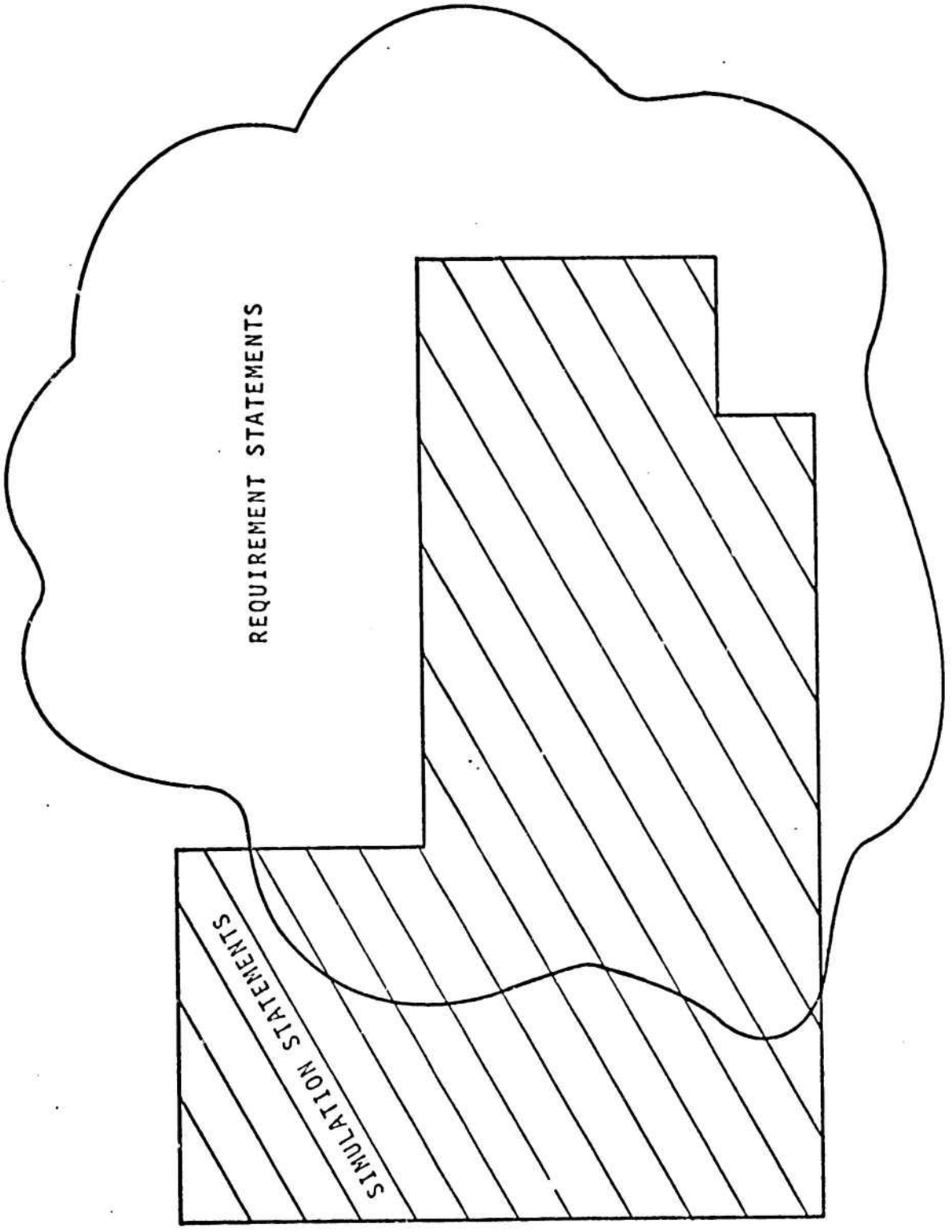


FIGURE 3-1 THE RELATION OF SIMULATION AND REQUIREMENTS

In summary, computer languages have generally emphasized procedural aspects of communication with less regard for non-procedural elements. However, due to the advances in language construction methods (e.g., meta compilers, translators, and macro processors) a high level language can easily be designed and implemented to meet almost any well defined specification. The major program remains with the accurate determination of existing deficiencies and the assessment of desired features. Toward this end Section 3.3 suggests guidelines for designing a language. In order to obtain a clearer understanding of the capabilities offered by languages available today, several have been selected and classified in Section 3.2. A general description of each language is given in Section 3.4. A detailed assessment of the languages in terms of the properties listed in Section 3.3 is given in Table 3.2.

3.2 EXAMPLE LANGUAGES

According to Sammet [L.49], the twenty-year period between 1952 and 1972 saw the development of over two-hundred higher level computer languages. Of this group approximately one hundred and seventy were in use during 1972 in the United States alone. A list of one hundred and eighty-seven languages appearing in the literature between 1970 and 1974 is given by Sammet [L.54]. This collection is certainly not exhaustive (new languages are being proposed and implemented in great numbers), but it does contain languages representative of all major categories. For our assessment, nineteen samples have been taken from this list which span the major applicable categories. The languages are of types, simulation or non-simulation. The primary interest is in simulation, however, non-simulation languages were included to provide a different perspective in order to obtain highlights outside the simulation regime.

There are many ways to classify programming languages and we have adopted a manner appropriate for our purposes suggested by Kosy [L.37]. He classifies languages according to levels of power (see Figure 3-2). Generally, those languages in the outer levels have the "power" and more of those languages in the levels interior. The inner most Level A is characterized by the algebraic languages. In this area, we have three example languages, FORTRAN,ALGOL, and JOVIAL. The next Level B is represented by those languages featuring list processing, data structures, and report generation. To this level belongs four of our nineteen languages, viz. COBOL, APL, LISP, and PL/1. Languages which in addition to those features mentioned in Levels A and B offer also the rudiments of simulation capacity are classified in Level C. The languages SIMSCRIPT, SIMULA, SALSIM, and PROSE are examples selected which belong in this category. In the event a simulation language offers macros dealing with larger than primitive processes it is classified as a Level D language. In particular, if the macro processes deal with aspects of computer system modeling the language is classified Level E. We have selected Level D languages ASPOL, SPCL, GPSS, PCL, and PDL and Level E languages ECSS II and CSS.

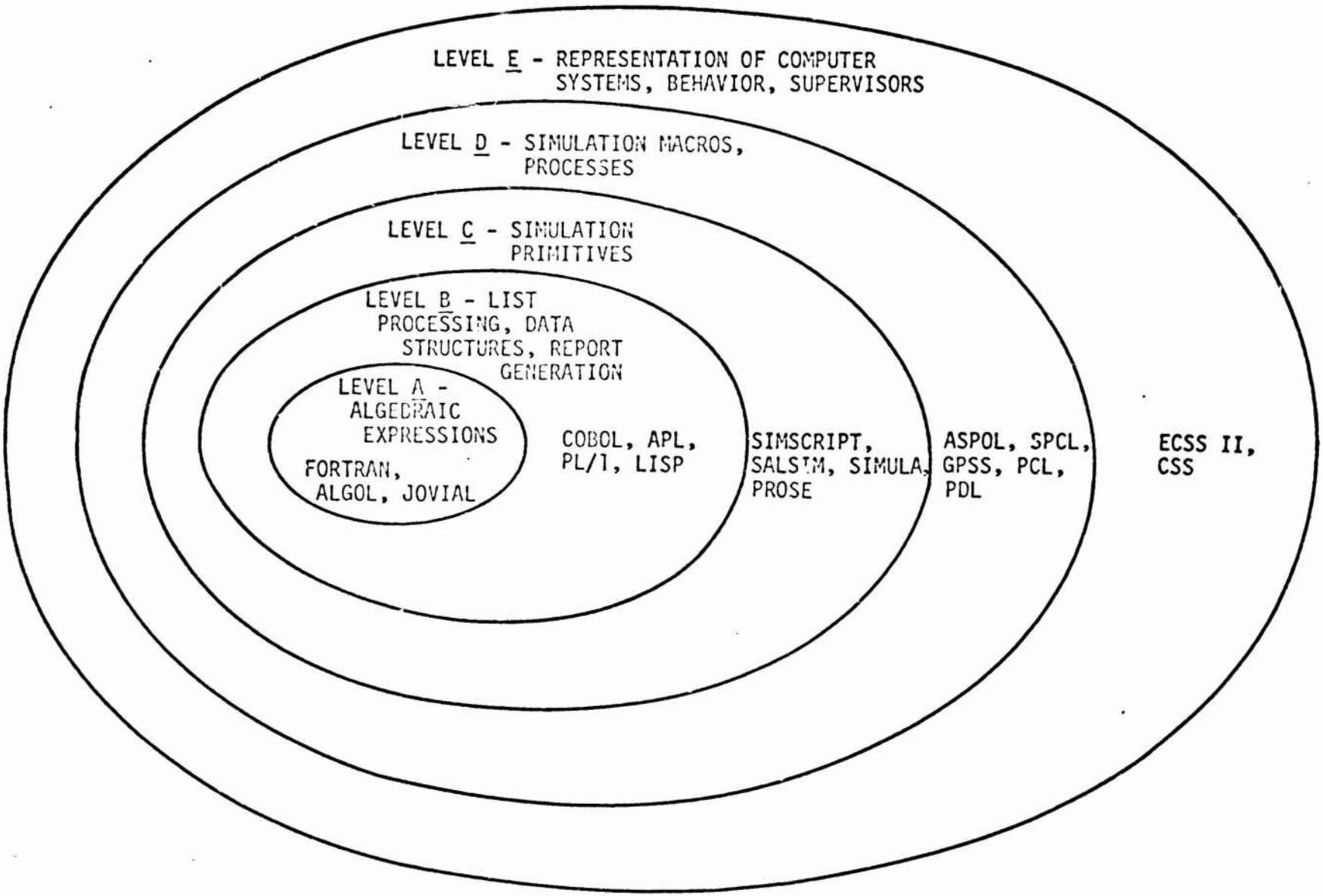


Figure 3-2 Levels of Language Power

3.3 HIGH LEVEL LANGUAGE REQUIREMENTS

Desirable language characteristics are enumerated here with a view toward the requirement statement and simulation specification relationship mentioned above. Some attention has been given to the language considerations of Kosy [L.37] such as availability, training required, portability, and vendor support. In addition, the checklist provided by Kiviat [L.34] has been consulted which includes such items as data collection and specification, display formats, data analysis, and others.

The list of desirable features (Table 3.1) consists of eleven categories: data reduction and report generation (DRRG), simulator execution, simulator construction, function modeling, readability/usability, data base control, library management, requirements specification, testing, documentation, and management visibility reports. Each category is further broken down into sub-categories detailing the features considered necessary for the performance of tasks in that area. To avoid ambiguities, the following paragraphs present, briefly, the scope of each subcategory.

Under the first category, data reduction and report generation, the requirements are for:

- A data collection facility to provide the user with a means of specifying, at one time for any given run, the data to be collected and the form of the output.
- I/O formatting to be implied from the data description or, optionally, specified by the user.
- Post-Processing of collected data to be done as specified by the user. It should include histographic and graphic display capabilities as well as tabulated data.
- A report generation capability to produce reports on program performance, computer usage, etc., upon request.
- A debug capability to be activated, deactivated, and delimited by the user.

The category of simulator execution is to include facilities for:

- Initialization of the data set by some method to allow preset data to be varied from run to run.

TABLE 3.1 HIGH LEVEL LANGUAGE REQUIREMENTS

DATA REDUCTION AND REPORT GENERATION

- Data Collection
  - Processing Macros
  - Recording Commands
- I/O Formatting
- Post Processing
- Report Generation
- Debug

SIMULATOR EXECUTION

- Data Set Initialization
- Test Case Selection
- Execution Job Step Control
- Exogenous Event Selection

SIMULATOR CONSTRUCTION

- Separation of Computer Independent and Computer Dependent Aspects
- H/W Config. Spec.
- Sys. S/W Config. Spec.
- S/W Module Selection
- OVLY Structuring
- Execution Control
- Time Modeling

FUNCTION MODELING

- User Defined Macros
- Executable Code (Arith & logical)
- "Data Set" Access
- "List Processing"
- Table Modeling
- Event Processing
- Time Posting

READABILITY/USABILITY

- Structured Prog. Conventions (for procedural specifications)
- Short Form/Long Form Expansion
- Interactive when possible
- Identification, Reference, Retrieval by User Assigned Names

DATA BASE CONTROL

- Global DB Dictionary
  - Elements
  - Grouping
  - Indexing
  - Ownership
  - Coord. Sys/Units/Type/etc.
- "Function" I/O
- Functional/Analytic Data Conversion
- File Structuring Control

LIBRARY MANAGEMENT

- Creation, Update, Purge, ...
- Configuration Control
- Multiple, Hierarchical Libraries
- Security Controls

RQMTS SPECIFICATION

- System Requirements
  - Structured operating rules
  - System performance rqmts/ design goals
  - Assumptions List
- Miscellaneous
  - Design Constraints
  - Preamble data
- Interfaces
  - DP/External I/F's
  - Functional
  - Detailed

TESTING

- Test Type
  - "System" Test
- Mode
  - Static Validation
  - Functional
  - Analytic
- Test Case Generalization/Selection
- Test Invocation
- Test Report Level (Type)
- Driver Selection

TABLE 3.1 HIGH LEVEL LANGUAGE REQUIREMENTS (CONTINUED)

DOCUMENTATION GENERATION	MANAGEMENT VISIBILITY REPORTS
• Media - Flow Charts - Graphics - Structured Text - I/O Devices • Type - Specification - Test Results - I/O Diagrams • Configuration Mgmt Controls	• Critical Path Analysis • Cost/Schedule • Resources • Responsible Party • Traceability/Design Breakage • Status • Error Reporting/Analysis

- Selection of test cases by name (or similar identifier) from a library.
- Control of job step execution order such that steps may be executed singly or in various configurations at the user's option.
- Admission of exogenous events into the system at any given time.

The simulator construction activity should be responsive to specifications of:

- Computer independent (CI) as well as computer dependent (CD) elements and be collectively aware of the difference.
- The hardware configuration necessary for the execution of the given simulation (e.g., memory size, number of discs, etc.).
- The system software configuration such as the necessary compilers, libraries, etc.
- The selection of particular software modules (this presumes a modular program structure which will permit the addition, deletion and interchange of modules).
- Overlay (or segmentation) structure.
- When (and if) execution of the program is to be initiated.

At the function modeling level the capabilities desired are:

- The admission of user defined processing.
- The generation of executable code.
- Access to the problem data set whether by files or other means.
- A list processing facility that permits access to list members on an individual member basis by user-assigned names.
- A means to do table modeling, that is, modeling parameter generation via table look-up rather than the individual specification of same.
- Processing of the simulation on an event basis (necessary for discrete simulation).
- Time posting.

For readability, as well as usability, the requirements call for:

- Structured programming conventions to be followed. This includes use of the IF-THEN-ELSE, WHILE, SEQUENCE, UNTIL FOR and CASE structures, segmenting of code in modules, and use of indentation in printed code to indicate logical relationships.
- A provision to allow the use of shortened forms for programming and their automatic expansion for listing readability.
- Interactive capabilities where useful.
- Allowing the user to identify, reference and retrieve data by names he assigns.

To allow the user data base control it is deemed necessary to provide for:

- A global data base dictionary which will reference data on the element or group level, permit indexing, retain data characteristics such as "ownership" and "units expressed in".
- I/O at a function level.
- A capability such that different representations of data needed for different simulation types (e.g., functional/analytical) will be supplied in a transparent manner once the type of simulation has been specified.
- File structure control by the user.

Library management requires facilities for:

- The creation, updating, and purging of library contents.
- Configuration control.
- Admission of multiple libraries and a defined hierarchy of usage.
- Controls to provide security for the information contained so it may not be accessed, altered, or destroyed by an unauthorized user.

In the category of requirements specification, needs were identified in the areas of:

- System requirements which includes the detailing of structured operating rules (a method of specifying system performance requirements as well as the design goals and a record of the assumptions under which the system is operating).

- A miscellany sub-category that will include means of imposing design constraints and defining preamble data.
- Specification of interface criteria, both function and detailed.

The area of testing has requirements for:

- Specification of the type of testing to be performed.
- Specification of the test mode: either static validation, functional, or analytical.
- A means for the generation/selection of a test case.
- A method of calling out testing when desired.
- Specification of the type and level of test reports.
- A means of selecting one particular test driver.

Documentation generation is broken down into needs for:

- User control over the documentation media.
- A method supplying user control over the type of documentation (specification lists, test results, etc.) necessary for a particular run.
- A method of configuration management.

The final category has been a most neglected area -- management visibility reports. Here the requirements identified are report capabilities in the areas of:

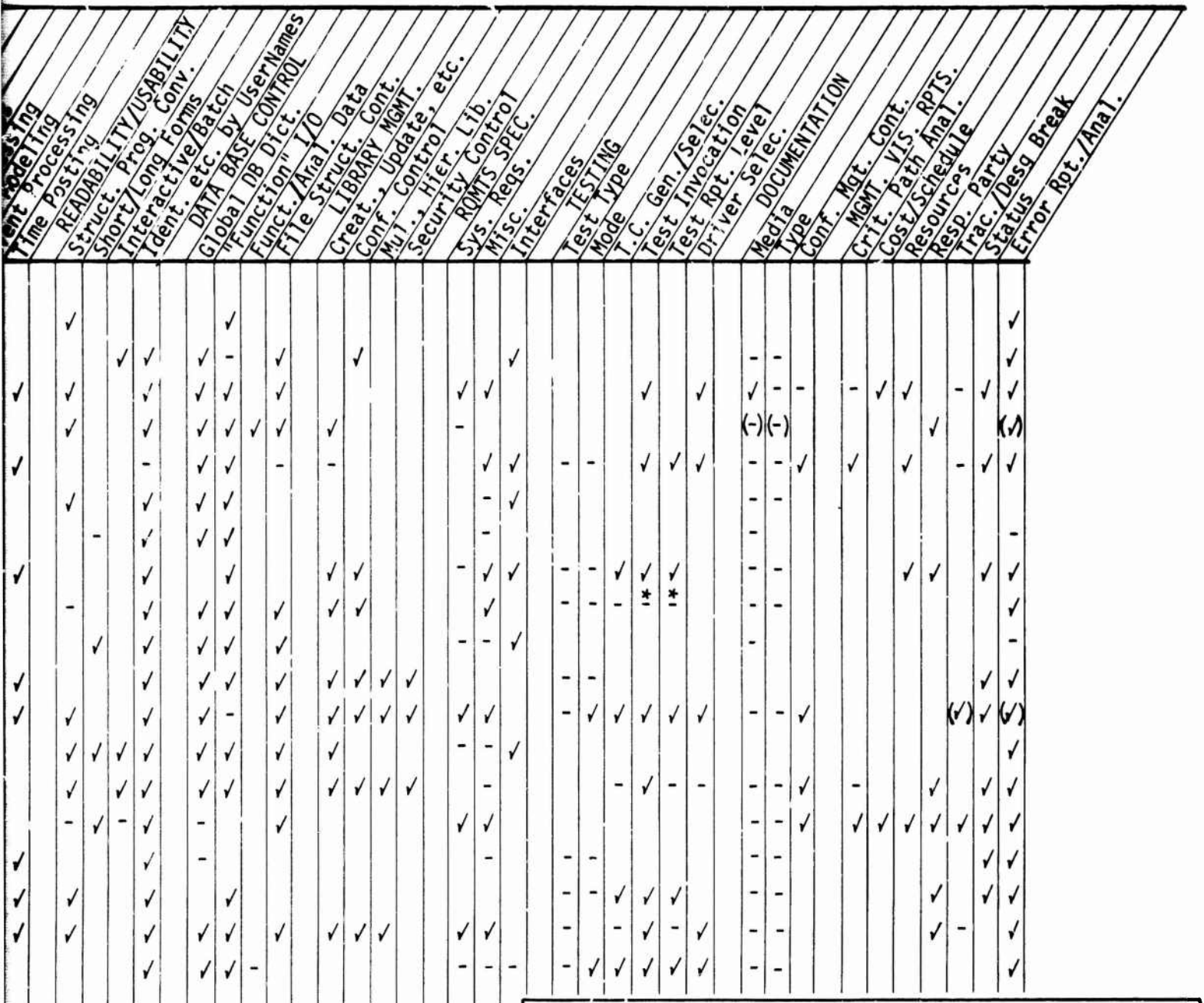
- Critical path analysis,
- Cost/schedule,
- Resources used,
- Responsible party,
- Traceability/design breakage,
- Status, and
- Error reporting and analysis.

Although an item such as "responsible party" is desirable output for each run, the balance of these are to be under user control. Table 3.2 presents the results of this survey as a matrix of languages versus requirements. When the distinction could be made between language and operating system capabilities only those attributable to the language itself were noted. A lack of sufficiently detailed information has resulted in incomplete data in some cases.

Table 3.2 Assessment of Lang

LANGUAGE \ REQUIREMENTS	DRRG	Data Collection	I/O Formatting	Post Processing	Rpt. Generation	Debug	SIMULATOR EXEC.	Data Set Init.	Test Case Sel.	Exec. Job Step Con.	Exogenous Event	SIMULATOR CNSTR.	CI CD Separation	H/W Config. Spec.	Sys. S/W Config. Spec.	S/W Module Set	OVLY Str.	Execution Control	Time Modeling	FUNCTION MODELING	User Def. Macros	Executable Code	Data Set Access	List Processing	Table Modeling	Event Processing	Time Posting	READABILITY/USABILITY	Struct. Prog. Conv.	Short/Long Forms	Interactive/Ba	Ident. etc.	DATA	Glo
ALGOL		✓					✓							–	–				✓	✓	✓	✓					✓						
APL					✓		✓			–			–	✓	✓		✓			✓	–	–							✓	✓			✓	
ASPOL	✓		✓	✓	✓		✓	–	✓				✓	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓		✓			✓			✓	
COBOL	(-)	✓		✓	(-)		✓		–			✓	–	–	✓	✓	–			✓	✓	✓					✓			✓			✓	
CSS	✓	✓	✓	✓	✓		✓	✓					✓	–	–			✓		✓	✓			✓	✓					–			✓	
ECSS			✓	✓	–		✓	✓	✓	✓			✓	✓	✓		✓	✓	✓	✓							✓			✓			✓	
FORTRAN IV		✓					✓												✓	✓	✓								–				✓	
GPSS/360	✓	✓	✓	✓	✓		✓	✓		✓				✓	✓			✓	✓	✓	✓	✓	✓	✓	✓					✓			✓	
JOVIAL		✓			✓		✓	*	✓	–				–	✓		–		✓	✓	✓	✓	✓	✓			–			✓			✓	
LISP		✓			✓		✓			✓					✓				✓	✓	✓	✓	–					✓			✓			✓	
PCL	–				✓		✓		✓						✓		✓	✓		✓	✓	–		✓	✓						✓			✓	
PDL	✓	✓	✓		✓		✓	✓	✓	–				✓	✓		✓	✓	✓	✓	✓	✓	–	✓	✓		✓			✓			✓	
PL/1		✓			✓		✓		✓	✓					✓		✓		✓	✓	✓	✓		✓			✓	✓	✓	✓			✓	
PROSE	✓	✓	✓	✓	✓		✓	–	✓	–				–	✓		✓	✓		✓	✓	✓	–	✓			✓		✓	✓			✓	
PSL/PSA (3.0)	–	✓	–	✓	✓		✓							✓	✓		–			✓	✓						✓	✓	–	✓			–	
SALSIM	✓		✓		✓		✓			✓					✓			✓		✓	✓	✓		✓	✓					✓			–	
SIMSCRIPT	✓	✓	✓	✓	–		✓	✓		✓					✓			✓		✓	✓		✓	✓	✓		✓			✓			✓	
SIMULA	✓	–	✓		✓		✓	✓	✓	✓				✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓		✓			✓			✓	
SPCL	✓	✓		–	✓		✓	✓	✓	✓			✓	✓	✓		✓	✓	✓	–	✓	–		✓						✓			✓	

Assessment of Language Versus Requirements



Key

✓	Meets requirements adequately
-	Meets requirements partially or potentially
Blank	Not provided for/missing
*	Tests may be done at parameter level

3.4 LANGUAGE DESCRIPTIONS

This section contains a brief summary and discussion of each language surveyed listing the notable features and any apparent advantages/disadvantages.

3.4.1 ALGOL (ALGOrithmic Language)

The history of ALGOL has been well documented [L.52]. As IAL (International Algebraic Language) its main intent was to provide a standardized language. In this connection, it was notable for the introduction of a language concept that had three specific levels: reference, hardware, and publications.

The reference language is computer independent and uses basic ALGOL symbols, such as begin and end to define syntax and semantics. The hardware language is the representation of the reference language symbols in a form acceptable to a particular computer. The publication language is one which interprets the reference language in a form suitable for printing. The formalized method of defining syntax developed by John Backus has become one of the more important contributions resulting from ALGOL. As a means for describing algorithms, ALGOL rates high but many of its other features are correspondingly poor. Because it was intended to be a machine-independent language no provisions were made for such features as: arithmetic precision, double precision arithmetic, handling of alphanumeric data or manipulation of complicated data structures. Most of these features together with input/output facilities relate to hardware characteristics and as such were left to be implemented by specific compilers.

ALGOL was designed for numerical mathematical and certain logical processes (e.g., sorting). Some original concepts it introduced included:

- three language levels - reference, hardware, and publication,
- Backus notation for syntax definition, and
- a block structure which allows unlimited levels of nesting.

An advantage of this language is that:

- it has the ability to define/express algorithms for a wide class of problems in a publication form.

Disadvantages existing are that:

- variations among implementation are considerable due to the subset of ALGOL used plus additions of machine dependent capabilities not included in the language definition (most of this work apparently has been done in error processing, debug, library, storage allocation/ segmentation areas), and
- data declarations are not as sophisticated as COBOL.

3.4.2 APL (A Programming Language)

According to its developer, Iverson ([L.20], [L.31], [L.35]), APL is a "...language...based on a consistent unification and extension of existing mathematical notations, and upon a systematic extension of a small set of basic arithmetic and logical operations to vectors, matrices, and trees. Unfortunately, the first thing many associate with APL is its exceedingly long (88 characters) and difficult to use character set. This does present a very real implementation problem since utilization of the full set currently depends on having a terminal equipped with an IBM Selectric typewriter which can use a special APL typeball. One solution to this problem has been to use a two character transliteration scheme.

APL is a very powerful language having single operators capable of performing number base conversions, matrix rotations, and identity matrix generations to name a few. An example of its power applied to data handling would be the deletion of a given break character from a data string and packing of the string all performed by a single statement. The problems inherent in such simplicity of programming are illustrated by McCraken's[L.44] statement in reference to a four-line matrix inversion problem: "It is a beautiful illustration of the action of quite a number of advanced APL operators, and for all I know it makes efficient use of the machine--but it took me four hours to figure out how it worked." This has not been too great a deterent, however, for many universities have found APL computer work considerably easier and, in 1970, the Atlanta public school systems was teaching APL to high school students.

One reason for this type of usage is the fact that APL can be learned and used a bit at a time-there is no need to employ the more advanced features or even know about them to start using APL. The essentials are easy to use. One definite plus in this connection is the handling of error messages--they are meaningful which is not always the case with other systems.

Since the language was designed for interactive use, it has features that lend themselves to this purpose such as the lack of required data definitions. There are only two types of data, numeric and character, and changes in arrays, sizes, etc., may be dynamically changed at execution time. There are also provisions for program editing one line-at-a-time interactively. A significant drawback, however, is the lack of any means to edit the entire program, e.g., using one program to create/edit another.

The absence of any static block structure (e.g., one program cannot be part of another although it may call another) could be considered another fault since this prevents the nesting of program blocks. The only method of going from one procedure to another is through a normal exit from the one and a return to where the call was executed in the other.

The lack of precedence rules for operators can be either good or bad depending on the user's viewpoint. As McCraken points out, initial reactions to an interpretation of 2 x 3 + 1 as 2 times the sum 3 + 1 is rejection on the grounds of "unnaturalness". However, the precedence list for PL/1 has nine levels to keep straight so this form of "naturalness" is also open to dispute. In either case, it would seem to be only a problem of becoming accustomed to the technique used.

As with most of the languages surveyed, the implementation of APL greatly controls its usability. This language is still evolving with one of the most recent implementations being made for Control Data Corporation's STAR (String Array) Processor. While at IBM Falkoff, Iverson et al are continuing with their work on APL.

Some advantages of this language are that:

- it is easy to start using as it can be worked into a bit at a time--it is not necessary to understand or know the whole language to use a given subset,
- it has good error handling, i.e., debugging supplies, meaningful messages,
- it usually has good response time (but may vary with implementation),
- there is good handling of vectors, matrices and other array problems,
- there are numerous powerful operators such as:
  - single operation number-base conversion,
  - single operation notation of matrix rows/columns,
  - single operation generation of identity matrix,
- it has been designed for interactive use, and
- it has no precedent rules for operators (+, -, /, etc.).

Disadvantages existing are that:

- present interpretations may be costly in terms of CPU time especially for certain number-crunching problems (this is an implementation related problem),
- the notation is difficult to read. (This seems to be a universal objection to APL.)
- there are no static block structure provisions--so that one program cannot be defined to be a part of another,
- generalized string operations are lacking,
- no provision has been made for operating on the program as a whole (although it may be edited on a line-by-line basis interactively), and
- the unusual and large (88) character set defies easy interpretation.

3.4.3 ASPOL (A Simulation Process Oriented Language)

ASPOL ([L.6], [L.39]) is a simulation language based on SOL together with process communication and control structures developed in computer operating system designs. ASPOL is included here as an example of features that can be implemented in a language of this type. The basic constructs are processes and events. A process is a dynamic entity and a system is considered to be a set of interactive processes. A simulation model would be

constructed from a set of process definitions. These activities are synchronized by means of events which may be either local or global in nature.

The major features include:

- the capability to concisely define systems composed of interactive processes,
- the capability to model parallel systems by definition and manipulation of sets of entities,
- simulation of many current operating systems simplified by an event structure resembling many of these current designs,
- inclusion of a macro feature allowing language extensions,
- data collection during simulation execution for facility, storage/queue usage, and
- structure based on processes and events.

3.4.4 COBOL (COmmon Business Oriented Language)

One of COBOL's most interesting features (see the articles [L.19], [L.23], [L.24] and the manuals [L.8], [L.59]) is the division of programs into the four segments of identification, environment, data, and procedure. This allows machine independence to a high degree even though data definitives may be influenced by the available hardware. The use of an English-like syntax was designed to allow program readability up through the management level. Although this goal may not have been totally achieved, this type of syntax made possible the use of COBOL by a relatively inexperienced programmer as well as provided a reasonably readable type of documentation for those other than the original programmer who might wish to examine the program. Unfortunately, this type of syntax requires a considerable amount of writing and so COBOL is often considered an anathema by the professional programmer. One attempt to overcome this problem was the development of RAPIDWRITE--a COBOL extension that permits the use of short forms with translation to full words at compile time.

As a language designed for business application, COBOL is a good massive data handler but its processing capabilities are rather simple. Although a self-extension capability was proposed, it was later dropped. The powerful number crunching available in other languages is unnecessary in business applications and so was omitted. Various other problems such as a limitation for batch use only, lack of error processing facilities, etc., have been provided for in various extensions and implementations. One of the most serious drawbacks and the one that tends to negate the ideal of machine independence are the variations encountered in the many implementations of the COBOL language.

In summary, some advantages of this language are that:

- it is computer-independent until execution time when the environment section must be added to the program,
- it is easy to read because of English-like syntax, and
- it has been designed for business usage and as such is a good massive data handler.

Disadvantages existing are that:

- the volume of writing required is objectionable to professional programmers (some extensions, e.g., RAPID WRITE, allow short forms),
- it is not a general language since it is oriented to solving business data problems,
- comments are permitted only in the Procedure Division, and
- there is a very large pool of reserve words that must be constantly checked while programming.

3.4.5 CSS (Computer Systems Simulator)

CSS ([L.26], [L.27]) was designed at IBM for the specific purpose of providing a means of simulating computer systems in a more specific and efficient manner than GPSS. It provides for computer system modeling through specification of the system configurations which draws from a library of elements that represent the actual computer components (e.g., disc, tape, etc.) rather than the method used in GPSS of modeling abstract equipment and micro operations.

In addition, the 'program' that will 'run' on the modeled system is treated as a separate entity and so its flow through the system can be specified. The major drawback to CSS is the fact of its limitation to IBM equipment exclusively.

Some advantages of this language are that:

- it has general equipment models incorporated into its basic structure--thus modeling consists of calling out the desired components,
- the program operation on a system being simulated is very close to the operation of the actual system,
- it can support a large number of system configurations, and
- the operations modeled may be sequential, multi-program, real-time or time-sharing.

Disadvantages existing are that:

- the components are IBM oriented (models System/360, System/370),
- it is not intended for general purpose usage,
- the user must be familiar with the working of OS (Operating System), and
- there are no automatic design features.

3.4.6 ECSS (Extendable Computer System Simulator)

This superset of SIMSCRIPT II was developed by RAND Corporation [L.38] for NASA with continuing work being supported by the Air Force. Although it is a special-purpose language intended for use in the simulation of computer systems it does embody many characteristics desirable in a general purpose language as well. These advantages as listed by Kosy [L.36] are the natural, English-like syntax, provisions for compact descriptions of computer elements/operations, flexibility, extendability, modifiability and the provision for economical program reruns (recompilation is not always necessary). At the same time, it has been noted that the report generation capabilities and data collection facilities are very poor. Other mechanisms are not as readily controllable or available as they should be. As a result of a study Kosy made, the following were concluded:

1) Procedural specification of a model is more flexible than declarative (albeit the declarative is more convenient),

2) Powerful statements are of great use in the development of course models but the finer the detail desired, the simpler the statements used (also the more numerous),

3) For adaptability, easy access to the service routine executive is needed, and

4) Preformatted reports on certain information of general interest should be available at the user's option.

In addition to these, no interactive capability is currently available and no consistency checking is done by ECSS.

Some advantages of this language are that:

- it has English-like syntax,
- hardware for the simulation is compactly defined,
- only single statements are required for storage requests, job starting, execution, data transmission,
- both flow-oriented (continuous) and event (discrete) types of systems can be modeled,
- no particular level of detail is required,
- it is easily extendable by SIMSCRIPT II statements or SIMSCRIPT II/ECSS statements combined,
- the modularity of ECSS service routines permit major changes to be easily accomplished,
- re-compilation is not required for all reruns, and
- it has been chosen by the Federal Simulation Center as its full simulation language.

Disadvantages existing are that:

- although all data is available, no automatic collection is done,
- no interactive capability has been provided,
- little to no self-checking (consistency) is done by ECSS,
- no file access modeling is done,
- there are no built-in polling operations (uses priority-interrupt mechanism),
- software overhead modeling is poor,

- degradation of execution rates is poor,
- the possibility exists that ECSS may over-perform unnecessarily,
- control program simulation interfaces need improvement,
- it requires SIMSCRIPT II, available on IBM and Honeywell machines,
- little documentation is available, and
- it is new, so there are bugs.

3.4.7 FORTRAN IV (FORmula TRANslation Standard)

Originally designed for scientific problem-solving, FORTRAN has become one of the most widely-used programming languages. Because it is so well known the discussion here will only be concerned with some of the extensions available. Those unfamiliar with the technical features of standard FORTRAN are referred to [L.43] in the bibliography. Its general availability has led to uses of FORTRAN never originally intended.

The spectrum of extensions range from the addition of macros/function routines to new translators. Carleton, Lego and Suarey [L.5] developed their Proposal Writing Language through the addition of statements such as PREPARE PARAGRAPH. FORMAC [L.55] was developed as an extension to FORTRAN to provide a formal algebraic capability based on an already existing numeric mathematical language. (This particular extension was later applied to PL/1). DSL/90 [L.61] introduced functional blocks, switching functions and function generators into FORTRAN. It allowed for alteration of input statement sequences and considered any operational statement properly sequenced if its inputs were all available. Graf [L.25] added graphic display capabilities to FORTRAN through extensions such as the display functions POINT, LINE, etc.

The range of problems covered by these few extensions, from simulation to proposal writing, give an indication of the scope of applications for which FORTRAN is used as well as its flexibility. At the same time, one of the major drawbacks of any language at this level is clearly illustrated...insufficient capabilities are available for sophisticated applications. In short, FORTRAN "...was designed so early, better ways have been found to do almost everything that is currently in FORTRAN." [L.52]

Some of its significant extensions include:

- a proposal writing capability,
- a graphics implementation,
- a formal algebraic manipulator, and
- a simulation facility based on block diagrams.

Some advantages of this language are that it is:

- well-known,
- available on most computers, and
- relatively easy to learn.

Disadvantages existing are that:

- it is error-prone in usage, i.e., many details must be considered by the user, and
- extensions are needed for most applications outside of arithmetic programming.

3.4.8 GPSS/360 (General Purpose Simulation System)

GPSS [L.28] was one of the earliest simulation languages and was intended for general applications. GPSS supports the modeling of hardware components on a primitive level (micro operations) of switches, facilities, etc. This is a source of inefficiency when the problem is complex but does permit the simulation of a wide variety of systems from steel mills to telephone exchanges. The programs are based on block diagrams representing the system to be simulated. A set of subroutines is associated with each of the standard block types. The system is represented in terms of these blocks, the program then creates transactions that are moved to the proper block where the actions associated with that block are executed. Although this feature may simplify the learning of GPSS as opposed to a statement-oriented simulation language, it does not have the flexibility of such a statement language.

An advantage of this language is that:

- it allows generalized hardware modeling on the level of primitive equipment types ("facilities", "stores", "switches") and operations ("blocks").

Disadvantages existing are that:

- any attempt to achieve greater detail in computer component modeling results in inefficiencies due to the space and time required to define such complexity, and
- the 'Program' running on simulated hardware is not treated as a separate entity.

3.4.9 JOVIAL - (Jules Own Version of the International Algebraic Language)

Originally based on ALGOL, JOVIAL [L.52] has undergone many modifications and much of the resemblance has been lost. This is a very powerful language with many unique provisions such as the partial work (both bit and byte) manipulations allowed, the capability of accepting machine code as well as other language code, and the packing at different levels (none, medium, dense), to name a few. Because of its great power JOVIAL is very difficult to learn. In addition, it has a definite lack of automatic report generation/documentation facilities. The user must provide all such routines. However, if proper usage is made of the facility permitting the free distribution of comments throughout the code a certain level of documentation may be achieved. Although intended as a general purpose, procedure-oriented language, JOVIAL has no provisions for time modeling/posting and thus would require extensions for simulation work. It was designed to handle large command and control structures as well as to solve large, complex information processing problems.

Some advantages of this language are that:

- it is a powerful, procedure-oriented, problem-oriented and problem defining language,
- subprograms may be independently compiled,
- subprograms in other languages/machine code are permitted,
- it uses a COMPOOL for system data and subprogram declarations,
- there are no formatting restrictions on source language,
- it has numerous file and table structures, and
- partial word manipulation is allowed at both bit and byte level.

Disadvantages existing are that:

- it is difficult to learn--being designed for use by professional programmers only,
- there is no time facility,
- no attention has been paid to management information type reports,
- any documentation/data collection wanted must be coded by user, and
- interactive capabilities are not in all implementations.

3.4.10 LISP (LISt Processing Language)

LISP [L.42] is one of several list processing languages. This concept was first introduced as an aid for the proof of theorems in the propositional calculus. Other areas that need this capability are compiler writing, manipulation of formal algebraic expressions, some types of linguistic data processing and much of the work in artificial intelligence. Because of the difficult notation used in LISP 1.5, an SL (Source Language) was introduced in LISP 2 which strongly resembles ALGOL. At the IL (Intermediate Language) level the syntax remained very much like LISP 1.5 and so little advantage was gained for the user since he must work at this level. LISP is a language for professional use only. It was designed to operate in batch although on-line versions have been developed. Language self-extension capabilities permit many drawbacks to be eliminated but no provision is made in the language for storage allocation so this must be provided for during implementation. The basic operation of LISP is controlled by function notation e.g., car, con, cdr often written abbreviated in a form like caddadaadar. Needless to say, a succession of these would not be easy to decode.

Some advantages of this language are that:

- recursive processes are allowed (this may not always be considered an advantage),
- it has a "Push-Pop" stack capability,
- LISP deals well with many non-numeric applications such as pattern recognition, information retrieval, etc., and
- it has been designed for use in a time-sharing environment.

Disadvantages existing are that:

- the design limits it for use by professional programmers only,
- polish notation is used for all algebraic manipulations (this may not be a disadvantage), and
- the extensive use of parentheses makes notation error prone.

3.4.11 PCL (Process Control Language)

PCL [L.64] was developed by TRW Systems and designed for use under the Process Control Program (PCP) to generate bound processes acceptable to TOS (Tactical Operating System) or FTOS (Functional Tactical Operating System) for the Site Defense Project. (The version surveyed interfaced with FTOS only). It is an extension of FORTRAN, using a statement syntax very similar to FORTRAN in which PCL and FORTRAN statements may (and usually are) intermixed. Additional capabilities (such as "List Processing") are scheduled for later versions. A second program (ADHOC) is required to interpret the data collection output. Since this output is formatted more along the lines of debug print, it is not particularly suitable for general purpose usage.

Some advantages of this language are that:

- it is a superset of FORTRAN and
- it provides many of the facilities needed for discrete-event simulation using process construction techniques.

Disadvantages existing are that:

- not all the facilities are implemented,
- its debug capabilities are weak, and
- because of being designed for a specific application, it has restrictions that would not be required in general.

3.4.12 PDL (Process Design Language)

PDL has been developed by Texas Instruments as the implementation portion of their Process Design System (PDS)... "an integrated set of software tools used by process designers to design and implement real-time software systems". [L.62]

PDL statements are used for describing the system to be simulated. Execution, under PDS control, takes place in five steps:

- Translation of PDL language statements into a FORTRAN configuration of the experiment,
- Computation of the FORTRAN code generated,
- Construction of the experiment from specified modules,
- Testing, through execution of the model specified, and
- Reporting the results by inputting collected data to user-supplied analysis and report programs (at present, this step is a stand-alone job).

It has been designed specifically for use on TI's Advanced Scientific Computer (ASC), a vector processor. Several of its more powerful features are designed to utilize the ASC's pipeline design.

Some advantages of this language are that:

- it models both system and external environment,
- has interactive capabilities, and
- is one of the few to incorporate configuration management capabilities.

Disadvantages existing are that:

- it has been designed for use on Texas Instruments' ASC (Advanced Scientific Computer) only, and
- all capabilities are not implemented.

3.4.13 PL/1 (Programming Language 1)

PL/1 ([L.15], [L.42], [L.48], [L.67]) is a general purpose language with a very wide scope of application. It has borrowed from several other languages, e.g., structure (like the PROCEDURE directive) and descriptive vocabulary from ALGOL; commercial data processing features from COBOL; some list processing features from LISP; and to a lesser degree some FORTRAN influence may also be found. This is not to say it is only a reiteration of other languages since many new concepts have been added such as in the lines of tasking and multi-processing. To permit one task to build a data structure for the use of another task, a single storage pool was developed which would be automatically freed only at the end of execution for the entire program.

Files may be opened in one task and closed in another; synchronization of parallel processes uses a 'lock' statement that is similar in operation to Dijkstra's semaphores.

There are problems however, like the one mentioned by MacLaren [L.40], the abnormal termination of a task can result in data being left undefined. The proposed standards for PL/1 have made several changes in the language specifications and improvements are being made on a continuing basis. In an attempt to ease the burden of long compile times resulting from the numerous features for user-aid, two compilers have been developed-one for use during checkout and another for optimization.

Originally named NPL (New Programming Language), PL/1 has a fair degree of machine independence. It is very general with a wide scope--a large and powerful language (intended as a replacement for both FORTRAN and COBOL). Being procedure oriented, it embodies most previous concepts in this line. No overlapping DO's are permitted. PL/1 is a programming and not algorithmic language (even though ALGOL concepts have been used) designed for mostly batch and multi-programming usage.

Some advantages of this language are that:

- data is available to all tasks from one pool and only destroyed if explicitly freed or at end of execution,
- a large number of facilities are provided for both compile and object time debugging and error checking,
- a file is available to all tasks,
- it uses "locks" (similar to semaphores) for the protection of share resources, and
- it possesses many features of COBOL, LISP, ALGOL, and some FORTRAN.

Disadvantages existing are that:

- the implementation has not been without numerous bugs,
- many of the features included to aid the user have resulted in inefficient compilation (this had the use of two compilers-checkout and optimizing),

- the language is too large,
- it has a large number of special cases, and
- graphics, formula manipulation, simulation and complicated pattern matching are not available (a disadvantage here since PL/1 is attempting to replace several languages).

3.4.14 PROSE

A very high level language ([L.13], [L.14], [L.63] based on SLANG, developed by PROSE, Inc., and currently available to Control Data Corporation, CYBERNET or SCOPE users only, PROSE is intended to be a complete general purpose language in the fullest sense. Its arithmetic capabilities span the range of simple arithmetic and algebra to vector/matrix algebra to such calculus operations as the automatic evaluation of first and second order analytic derivatives, evaluation of maxima and minima of functions, solutions of systems of equations, including both algebraic and ordinary differential equations (implicit/explicit, linear/non-linear) to name a few. Capabilities are also provided for the modeling of both discrete event and continuous systems. Language extension is allowed for through a general purpose macro processor while interactive capabilities are handled through a time sharing executive, PRIMP. (This includes both program synthesis and job processing applications.)

Although not providing all the capabilities desired (see Table 3.2) PROSE ranks very high among the languages surveyed. Perhaps its biggest drawback is the fact it is still under implementation (the simulation capabilities were scheduled for release September 1974) and thus is unproven. The mathematical capabilities are superior and have been in use for a while with good reports.

Some advantages of this language are that it has:

- wide range of capabilities ranging from simple arithmetic through vector/matrix algebra to calculus operations,
- simulation capabilities for both discrete and continuous system modeling,
- extension capabilities, and
- both batch and conversational time sharing execution.

Disadvantages existing are that:

- it is currently available only to Control Data Corporation CYBERNET/SCOPE users, and
- it is still in implementation stage (simulation capabilities scheduled for September, 1974).

3.4.15 PSL/PSA (3.0) (Problem Statement Language/Problem Statement Analyzer)

Developed by the University of Michigan's Project ISDOS [M.25], PSL/PSA was intended to be used in that portion of a system concerned with the development and specification of requirements. The first part, PSL (Problem Statement Language), is a language for describing the objects in an information processing system and their relationships within that system. Statements in PSL are input to PSA (Problem Statement Analyzer), a software package which processes these design statements and provides complete documentation of the system requirements. No object code is produced at this time. The documentation output is intended to be used as input to the design and construction phases of the system. This language is included here primarily as an illustration of the automation available for requirements specification.

An advantage of this language is that it provides:

- a concise framework for system definition and
- complete documentation of system requirements.

Disadvantages existing are that:

- it is not a complete language since no object code is produced, and
- it is business data processing oriented.

3.4.16 SALSIM (Systems Analysis Language for SIMulation)

SALSIM [L.65] is a collection of FORTRAN IV program packages developed by TRW Systems. Discrete-event simulation systems can be built from flowcharts expressed in the Logic Control Chain (LCC) language together with simulation routines provided by SALSIM and any FORTRAN code required. Although the basic routines are functionally similar to those provided by SIMSCRIPT they require

less memory and execute faster. The LCC's are converted to FORTRAN IV subroutines by simple rules which maintain a one-to-one correspondence between the flowcharts and the program. A system model, system environmental model, and a reporting mechanism are constructed by the LCC's and FORTRAN. The SALSIM Executive controls the simulated time with data cards being used for any external imputs to the system. This combination provides the capabilities of SIMSCRIPT and GPSS, requires less memory, operates faster and is easier to learn and apply. SALSIM has been operational since 1968 and is currently used on several IBM and Control Data machines.

The major advantages are:

- it duplicates many features of SIMSCRIPT,
- it improves many features of SIMSCRIPT
  - faster execution times and
  - requires less memory, and
- it is written in FORTRAN IV providing
  - broad availability and
  - easy understanding and usage by the FORTRAN community.

The major disadvantages are:

- it is general purpose so there is no specific facility for applications modeling,
- extensions are needed for most applications, and
- many details must be provided by the user.

3.4.17 SIMSCRIPT

Although classified as a simulation language, SIMSCRIPT [L.41] possesses enough of the features found in scientific programming languages to be considered an effective tool for general purpose usage. It is very FORTRAN-like in appearance which simplifies the learning process for most FORTRAN users.

SIMSCRIPT has been significantly improved over earlier versions. The method of specifying entities and attributes, making storage allocations, and

specifying general purpose capabilities have been up-graded and recursive subroutine calls are now permitted. It has not been provided with interactive capabilities as yet and execution speeds may be slow in complex problems. Nonetheless it is one of the most powerful simulation languages having a wide range of application. This language was developed by RAND Corporation for simulation work.

Some advantages of this language are that:

- it has an instruction repertoire similar to FORTRAN,
- it is available on a variety of computer systems,
- the concept of entities having associated values that are subject to periodic changes is used,
- many data-processing features as well as those of a scientific programming language are incorporated,
- many automatically defined variables (e.g., current page number of printer output) are available to the user.

Disadvantages existing are that:

- no facility for modeling hardware components is available,
- no interactive facility has been provided for at present,
- execution times may be slow.

3.4.18 SIMULA (SIMUlation LAnguage)

SIMULA ([L.11], [L.16], L.29]) is "a dynamic, general purpose, programming system for algorithmic applications with a powerful simulation capability." The implementation surveyed (CDC's) was based on the SIMULA 67 Common Base Language as defined by the Norwegian Computing Center, Oslo [L.17]. The base for SIMULA is ALGOL 60 and the CDC version compiler uses the design principles of the GIER ALGOL compiler together with the SIMULA implementation guide. However, some of the ALGOL features were not implemented (such as the own declarer). This is a powerful language in addition to the unlimited range of extensions permitted the user. The syntax is easily read and input/output is quite flexible and is easily adapted to varying problem simulations. I/O may be continuous or not during execution. Facilities for editing both data and text are provided.

Some advantages of this language are that:

- Input/Output is easily adaptated to different problem situations,
- unlimited language extension is provided the user,
- classes are used to describe changing (appearing/disappearing) activities,
- it has an easily readable (and printable) language,
- it is available on most machines,
- sequencing of events is performed by scheduling algorithms,
- list processing capabilities are good,
- general purpose programming capabilities as well as simulation are included, and
- it handles flow type problems thru process concept that is both an 'event subroutine' and an entity in the simulation.

Disadvantages existing are that:

- it is weak in area of statistical reporting, and
- the implementation surveyed (CDC) restricted ALGOL 60 reference language somewhat e.g., own declarer, not allowed.

3.4.19 SPCL (Simulation and Process Control Language)

A simulation language developed at System Development Corporation [L.45] SPCL includes FORTRAN and ENLODE (GRC) as subsets. Although intended only for use in discrete, time dependent simulation it has several features of interest. In particular, through the intermingling of SPCL, FORTRAN and ENLODE a gradual progression from a functional model to an analytical model may be made. It also permits the selective exclusion of portions of a process during the translation phase.

Some advantages of this language are that:

- the free intermixing of SPCL, FORTRAN and ENLODE language forms allows progression from a functional to analytical simulation for process constituents,
- combining of prestored process segments is allowed, and
- it provides for selective exclusion of portions of process during translation as well.

Disadvantages existing are that:

- implementation has been made for Control Data Corporation 7600 only and
- it is intended for discrete, time dependent simulations only.

4.0 TESTING OF REQUIREMENTS

This section is directed at the issue of testing and debugging software requirements. Due to the fact that the testing and debugging activity today is almost entirely in the software circles and not in software requirements the survey concentrated therefore on the former. However, the separation of these two topics may be only artificial. In fact, one might conjecture that with regard to testing (and possibly other attributes) the discipline of software requirements is just an isomorphic copy of software. In the following paragraphs, these two topics are addressed with this spirit in mind.

4.1 DISCUSSION

In order to scientifically address the issue of correctness, it is desirable that the object being addressed have a well defined (and, if possible, simple) structure. Then this structure can be systematically examined for correctness (i.e., completeness, consistency, integrity, etc.) against some nominal defined as correct. Otherwise, if the object is without structure and amorphous in nature then scientific methods seem to be generally inapplicable. With this in mind then a major step in the testing of software requirements is the identification of some structure. Many of the articles reviewed are directed at analyzing this issue. In this direction, it seems reasonable that any structure attributed to software requirements be similar to the structure of software. Moreover, one of the most important principles in testing software requirements is the use of simulation which is software. Therefore various disciplines and tools related to this problem in the software regime have been examined. Table 4.1 lists various items encountered and indicates the area of application with regard to structure and testing. In testing, validation refers to the correctness of software requirements with respect to system goals and objectives. Verification addresses the correctness of software with respect to its specifications. The partitioning of this list is entirely arbitrary and is not to imply non-overlap of any two items. On first examination of the table, some of the items may appear to be obliquely related to this subject matter, but it is felt that the material does or can impact directly

Table 4.1 Disciplines and Tools for Testing of Software Requirements

ITEM	EMPHASIS		
	STRUCTURE	SOFTWARE TESTING	
		VALIDATION	VERIFICATION
1. Information Algebra	H	L	L
2. Systematics	H	L	L
3. Formal Proofs	M	L	H
4. Structured Programming	H	L	M
5. PSL/PSA	M	M	M
6. Automated Tools	L	L	M
7. Simulation	L	M	M
8. Formal Languages	M	L	M
9. System Modularity	H	L	M
10. Top Down Programming	H	L	M
11. Langefors	H	M	L

Applicability

H - high
M - medium
L - low

the testing of requirements. In the next section, a short description is given for each of the items in Table 4.1.

Several references examined ([T.12], [T.23], and [T.26]) were review articles in testing, debugging, and software reliability. These articles provided an excellent overview of this environment.

The contents of [T.12] are based on proceedings of the Computer Program Test Methods Symposium held at the University of North Carolina, Chapel Hill, June 21-23, 1972. Hetzel [T.12] in summarizing the proceedings proposed a definitional framework for the testing and debugging environment (see Figure 4-1). The dotted rectangle in the diagram represents the testing activity which is distinct from the debugging activity below it.

The debugging activity starts with some known error then produces a fix to the program so that the program then operates and solves some problem correctly. However, whether the problem being solved is the one desired or not is not addressed by the debugging activity. This aspect is decided in testing. Thus, an exchange of information is depicted between debug and testing.

The testing activity starts with some assertions and specifications about program behavior. In the figure, the inputs to the boxes indicate the information used, the output indicates the information obtained. Five different areas in the testing environment are distinguished. Certification carries the connotation of authority and usually asserts a certain standard or quality, in writing. Formal proofs and verification is only concerned with a program's logical correctness while validation extends this concept with regard to the environment. Verification and performance testing base their conclusions on program executions whereas validation includes environment information.

In addition to this framework in [T.12] is contained an annotated bibliography of the testing literature. Reference is made also to a comprehensive literature review by M. R. Paige and E. F. Miller of General Research Corporation in Software Validation.

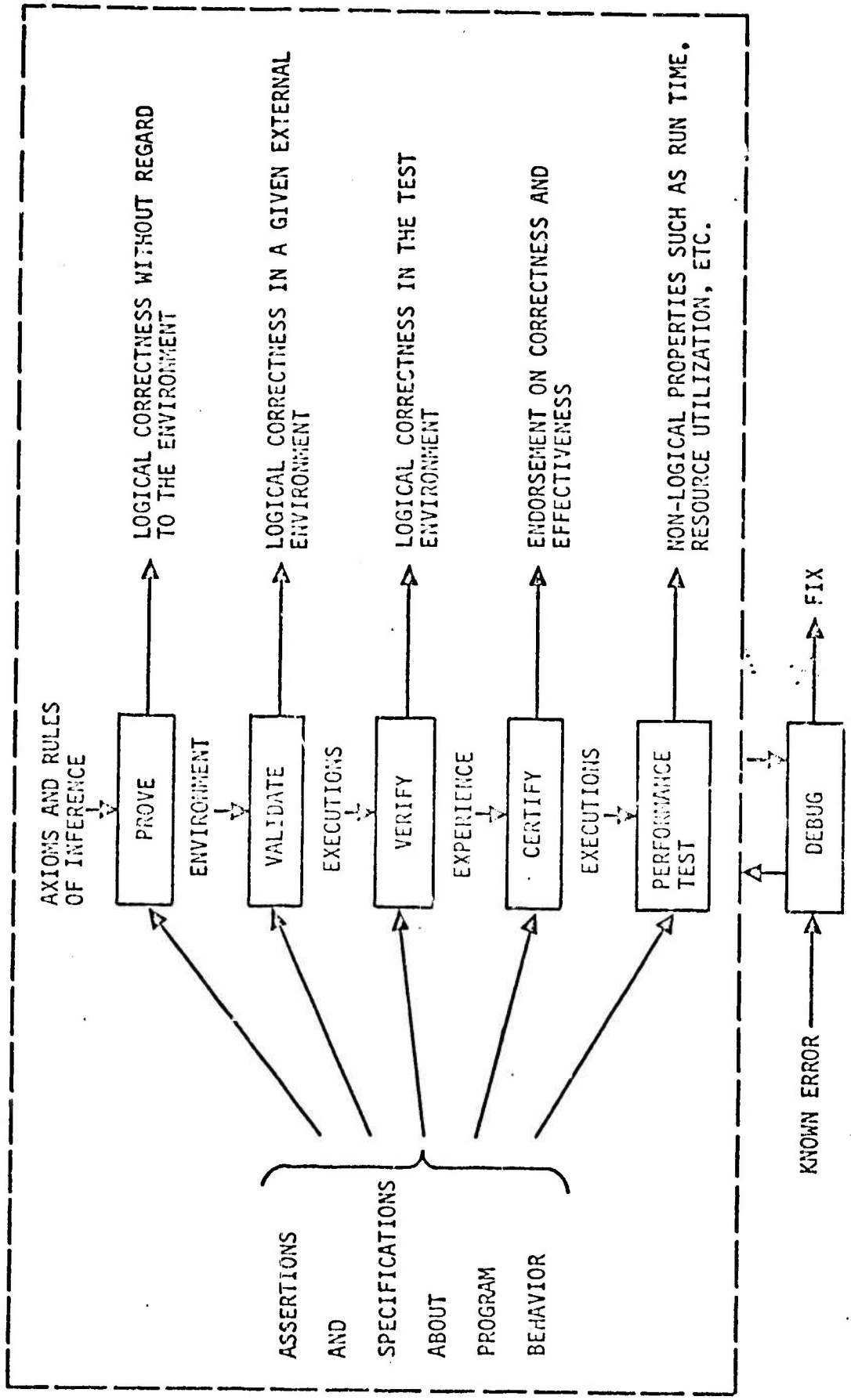


Figure 4-1 A Definitional Framework

Contained in [T.26] is a collection of papers given at the Courant Institute of Mathematical Sciences, New York University, June 29-July 1, 1970. The first paper "An Overview of Bugs" by Jacob I. Schwartz provides an overview of the text and a pointer toward further developments. Emphasis is placed on the debugging area, however, Mills' Top Down Programming" is in this volume as well as an article by Jim King on formal automated proofs.

The IEEE symposium [T.23] was a conference "to challenge the critical problem of software reliability from a combined practical and theoretical viewpoint". The papers contained therein are representative of the latest research in software testing. Unfortunately, the Panel Discussions are not contained.

Carey [T.3] in a review of the IEEE symposium, notes that six areas of software reliability were addressed: 1) system design, 2) testing techniques, 3) testing tools, 4) modelling of software reliability systems, 5) management techniques, and 6) programming techniques. Carey remarks that there was a total absence of papers on formal proofs concluding that this method is probably not practical to date for medium to large programs. In addition some doubt is expressed by Carey in a forecast of Harlin Mills who authored one of the papers at the symposium. Mills states that he has a programming approach which will propogate a new generation of programmers who will produce less than one software error per man year.

Generally speaking, we can conclude that software testing is little more than an embryonic activity but the disciplines and tools available now can be of good utility. However, in the requirements testing regime, it is unclear if there is much scientific activity. There are some positive steps being taken in requirements testing and these lie in the area of structuring requirements. The most notable effort seen is the PSL/PSA development activity of ISDOS [M.28] at the University of Michigan. The emphasis here is on a concise machine readable problem statement, documentation, data base design, and I/O checking.

4.2 DESCRIPTION OF VARIOUS DISCIPLINES AND TOOLS

4.2.1 Information Algebra

Information Algebra [T.4] is an attempt to obtain a proper structure for requirements of an information processing system. The primary discussion is about data and data relationships. Machine independence is an underscored characteristic of this discipline. The representation of data and data relationships is generalized and has been drawn from concepts of Modern Algebra and Point Set Theory.

4.2.2 Systematics

Systematics [T.10] is a language which systems analysts may use in designing information systems. The language is theoretical, without compiler, and computer-independent. The overall objective is to give precision to the stating of specifications of information systems. Mathematical concepts and notation are used and augmented by other symbology peculiar to information processing.

4.2.3 Formal Proofs

This category belongs to the area of proving the correctness of computer programs. The general idea is: given a program and a set of assertions about what the program does determine the validity or non-validity of the assertions. Generally speaking, this discipline is in its infancy, however, correctness proofs for programs of around one hundred statements have been completed. Most proofs in the final analysis are formal and require personal examination. Some effort toward automatic proofs are being examined.

It must be made clear that formal proofs are radically different than the other methods of testing. Verification and validation can prove that a program is incorrect. But as Dijkstra points out this activity cannot prove a program correct.

Linden [T.16] and London [T.17] give summary reviews of the state-of-the-art in this area. In Elspas et al [T.7] is a detailed technical overview of this area. Complete bibliographies may be found in each of these references. The mathematics for this discipline was originally worked out by Manna [T.18], [T.19] and Floyd [T.8] with the pioneering being supported by King [T.13], Maurer [T.21], Naur [T.24], and Rutledge [T.27].

There has been some emphasis on examining the literature in this area of testing. It is hoped that the methods for proving program correctness can be applied to questions involving requirements validation and thus assert correctness to some aspects of a program's relation to its environment.

4.2.4 Structured Programming

Structured programming [T.6] is a discipline introduced by E. W. Dijkstra in order to give sound framework and clarity to computer programs. The resulting program becomes much easier to understand and moreover verification and formal correctness proofs become more readily applied. Some of the structure is obtained by elimination of the "go to" statements to prevent circuitous paths. Many of the programming tips offered are very sound and in fact are just thinking principles and are of course applicable in the requirements determination phase of software development.

4.2.5 PSL/PSA

Problem Statement Language (PSL) [M.28] is a machine readable language for describing an information processing system. A problem statement in PSL can be used to describe the present system or to state requirements that a proposed target system is to fulfill. The software package, Problem Statement Analyzer (PSA), is intended to accomplish many of the clerical and analytical tasks which previously had to be carried out manually. These tools are generally accomplished by using computer-aided methods of arranging and displaying tables, arrays, matrices, diagrams, and flowcharts. Section 2.3.3 contains further elaboration on this subject.

4.2.6 Automated Tools

Hetzel [T.12] proposes that automated tools can be conveniently grouped into five categories:

- test data generators
- checkers
- testing estimators
- transformers
- monitors and simulators.

The test data generators are perhaps the earliest of the automated tools. These can be generally grouped into those independent of program logic and those dependent on program logic.

The checkers are those programs which examine module interfaces, data files, and relationships of program variables. Theorem provers can also be viewed as belonging to this category.

The testing estimators attempt to measure or estimate the degree of testing. This is accomplished by analyzing program execution and noting branches tested or not tested, percent of instructions executed, modules or subroutines entered, etc.

The transformers translate programs and program modules to equivalent forms in order to simplify the structure and for readability. Examples are the flowcharting tools and the digraph tree structure converters.

The monitors and simulators control the test environment. Some systems are built in order to provide easy simulation capability in various areas.

References [T.9] and [T.29] impact automated tools in all five categories and represent more general but less detailed viewpoints. References [T.1] and [T.2] belong to the testing estimators and monitors while [T.14] uses and contributes technology in the areas of test data generators, testing estimators, and transformers.

4.2.7 Simulation

The principle of simulation has become more and more important as systems have increased in size and complexity. Simulation's main usefulness to testing is its ability to provide a controlled program environment. This aspect has been an invaluable one to the testing regime.

However, two difficulties with simulation are prominent. First is the problem of insuring that the environment the simulator actually represents is in fact the desired environment. Second, the simulator software is often more difficult to test and debug than the target software.

5.0 MANAGEMENT TECHNIQUES

Even a cursory review of the available literature on the special management techniques for requirements engineering would point out its paucity. In fact, even the more general material concerning the management of software development lends little aid in determining effective management goals and techniques for this area.

5.1 SOFTWARE MANAGEMENT DATA

Weinwurm [M.26] performed a classical study in 1966 on the economic analysis of computer programming. This study was based on empirical data from 174 computer programming projects completed at industrial and Air Force organizations. This article remains the most prominent in the literature on economic analysis of computer programming. Weinwurm pointed out the difficulty of measuring empirically the cost of programming; these difficulties still hold today and include:

- "Computer programmers generally do not describe the process, the steps in the process, or the end products in comparable ways."
- "There are no generally accepted, comprehensive, and validated measures of computer program complexity or difficulty."
- "Even if adjustments could be made for the complexity and difficulty of the job, the absence of any satisfactory or meaningful measure of computer programming quality implies an additional uncertainty."
- "Assuming that both job difficulty and product quality can be taken into account, any experience-data that might be collected will embody a substantial additional bias that reflects the differing 'power' of the configured computing machines."

Weinwurm's study made a tentative step toward the derivation of performance measures and formal study of the economic process of programming.

The economic measures that were developed or used by Weinwurm included a computer usage rate, a production rate and the "Shaw Index." His computer usage rate is basically computer hours divided by machine instructions; the production rate is machine instructions divided by man-months. The "Shaw Index" is the high speed memory transfer rate in bits times Log_2 of bits in memory.

The set of programs upon which data were available was divided into four categories:

- Business applications--where file manipulations were the predominant feature of the program,
- Scientific--where extensive computations were performed on relatively small data inputs,
- Software--where the programs were designed to support programming and computer operations (largely operating system implementation),
- Research and Development--where the programming effort was novel.

A further division of these categories was made into two areas:

- Procedure-oriented languages (such as FORTRAN), and
- Machine-oriented languages (including assembly languages).

Weinwurm's results have certain limitations due to the restricted amount of data available. In fact, he was distressed that "the results have been so superficial." Reviewing the literature makes clear that adequate data has not been collected since that time, so his rather preliminary results must be used as the best available.

The available data led to the conclusion that different sorts of problems existed for the various kinds of systems being developed. For example, Weinwurm found that Research and Development (R&D) programs and "software" programs, taken together, were significantly different from the business and scientific programs, taken together. The R&D and software programs, from the empirical data, required significantly less computer resources in development than did business and scientific programs. This implies that emphasizing the differences between business and scientific programs may not be useful. However, Weinwurm's data do not include samples of real-time command and control systems nor of process control systems development. Therefore, we can only conjecture that conclusions drawn about the more conventional, first four, kinds also apply in the systems of particular interest in the BMD environments. This assumption appears justifiable if we associate real-time programs with the category of R&D and "software" programs.

Weinwurm found that there may be more state-of-the-art programming in research and development programs than is generally expected. This certainly appears consistent with experience in real-time software, as does his statement that "To the extent that this is generally true, it would help to explain the rather serious difficulties often encountered by those who have considered the development of software as more or less a production task and have managed accordingly." However, techniques for managing production efforts are rather well-developed while management techniques for developing software are rather rudimentary. Weinwurm recognized this problem and stated that "The main impediment in the way of a remedy is a tradition: that of emphasizing the forceful and deliberate evolution and application of technologies and relegating to a subordinate and often incidental role the concomitant development of particular tools of management, without which the technologies cannot serve economic and social ends reliably and well. As a result of this viewpoint, immense resources are customarily accorded to the former while relatively minuscule and evanescent investments are more often the case with regard to the latter. Computer programming (and the balance of information processing as well) has not been immune to this tradition." Recently, some effort has been devoted to managing the programming job (with the Chief Programmer Team) but no effort has been devoted to developing management techniques for obtaining meaningful requirements.

5.2 THE DEVELOPMENT CYCLES OF SOFTWARE PRODUCTS

It becomes apparent when reviewing the available literature on managing software projects that there is a different perception of the development cycle for projects. This is illustrated in Figures 5-1 and 5-2. Rubin in [M.25] presented Figure 5-1 to represent the system life cycle of a software project. The project begins with the conception of the system and then proceeds through six steps until it finally reaches the system cessation. Royce in [M.24] represented the waterfall chart shown in Figure 5-2 which represented the steps in a large scale software development project.

Both figures are broad overviews of the software development cycle. Consequently, the obvious differences such as a step called system documentation in Rubin's figure may be contained in one of the other steps in Royce's figure. What is important is the different emphasis placed on different aspects of the development cycle. Both Royce and Rubin place a high emphasis on the documentation. Consider the following statement made by Rubin.

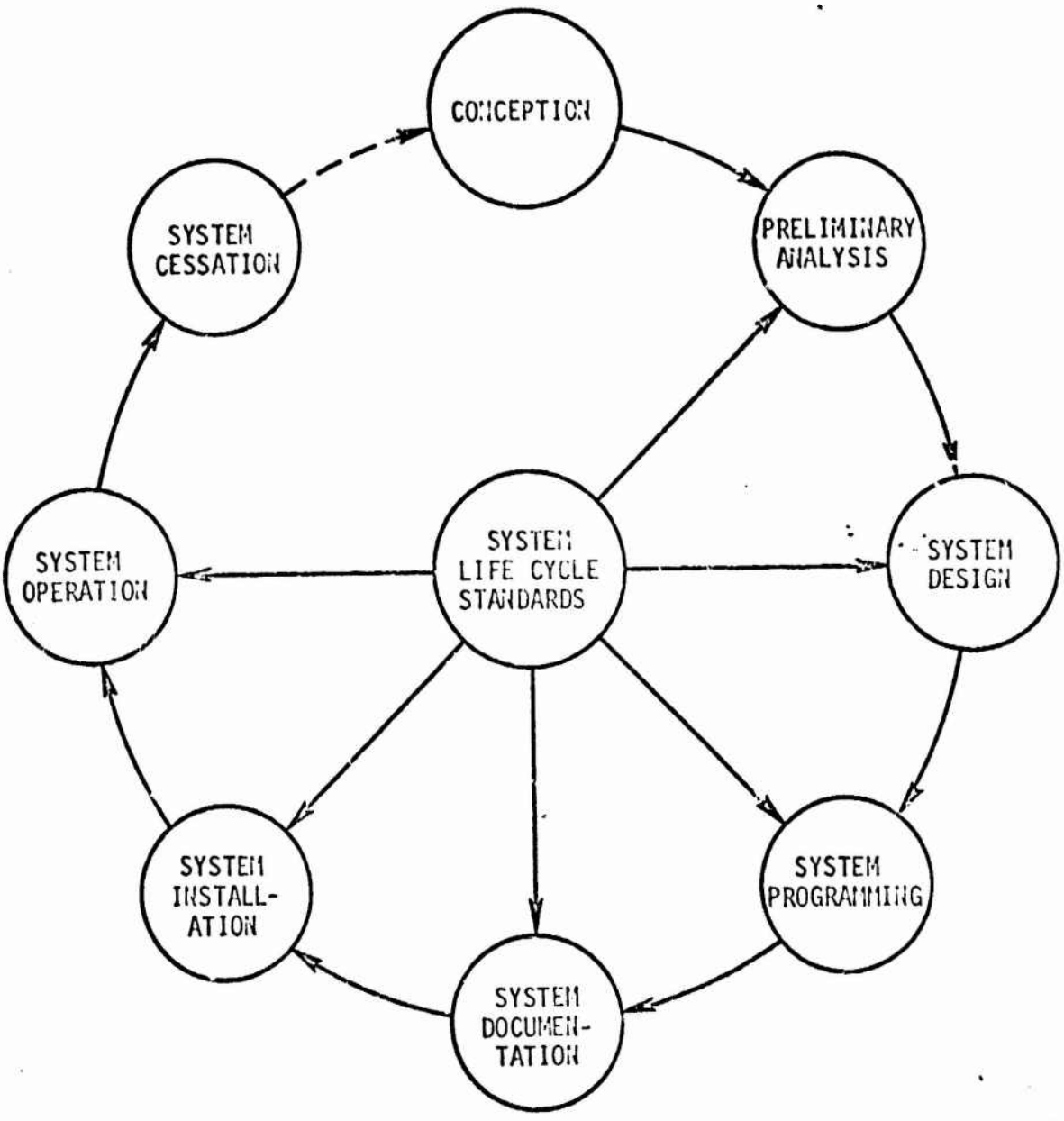


Figure 5-1 Steps in the Software Development Cycle (Rubin)

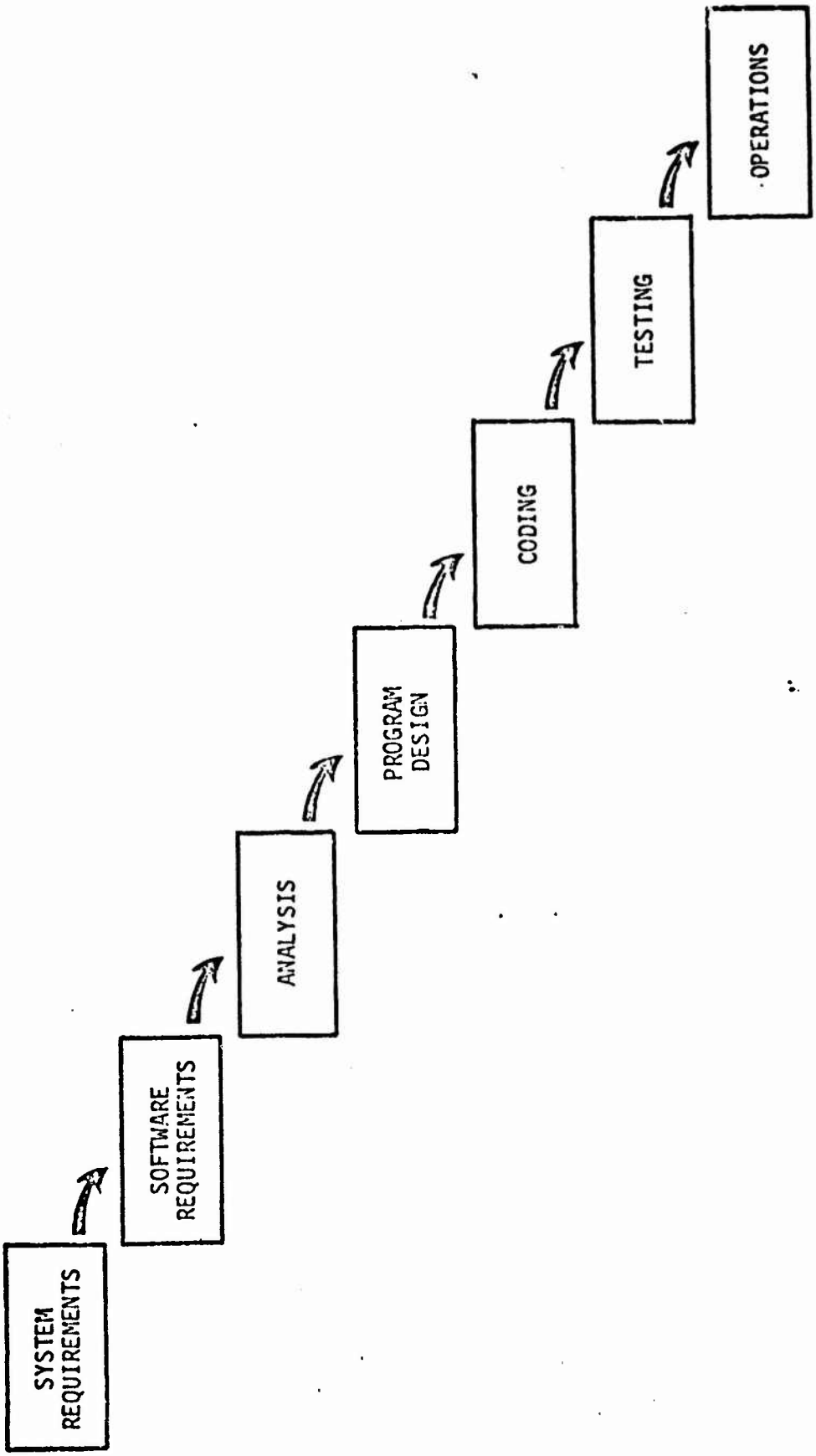


Figure 5-2 Steps in the Software Development Cycle (Royce)

"The System Life Cycle puts documentation where it belongs: equal in importance to analysis, design, and programming. More management effort must be exerted to obtain good documentation than is common to obtain good analysis, design, or programming because analysts and programmers, upon completion of those segments, feel that they have finished their assignments." Royce says "The first rule of managing software development is ruthless inforcement of documentation requirements."

Royce also emphasizes the feedback loops in the software development cycle in subsequent expansions of Figure 5-2. In addition, Royce adds a block to show the completion of a preliminary program design prior to commencing the analysis because "By this technique the program designer assumes that the software will not fail because of storage, timing, and data flux reasons. As the analysis proceeds in the succeeding phase the program designer must impose on the analyst, the storage timing, and operational constraints in such a way that he senses the consequences."

Rubin also suggests the following characteristics be attributed to the management of a software project:

- A definite set of goals
- A definite life
- A team that is disbanded upon completion of the project
- Authority and responsibility vested in a project manager
- Use of scientific management techniques, including a predetermination of functions
- Identification of milestones that separate development segments
- "Go-No Go" points in the life cycle where the option to drop the project can be exercised.

These attributes may be tempered if applied to a large scale real-time program but are still basically true.

Figure 5-3 shows the Terminal Defense Program (TDP) development process for research and development program leading to a real-time software system. This differs from those processes shown in Figures 5-1 and 5-2 leading to the conclusion that the nature of the software system being developed contributes

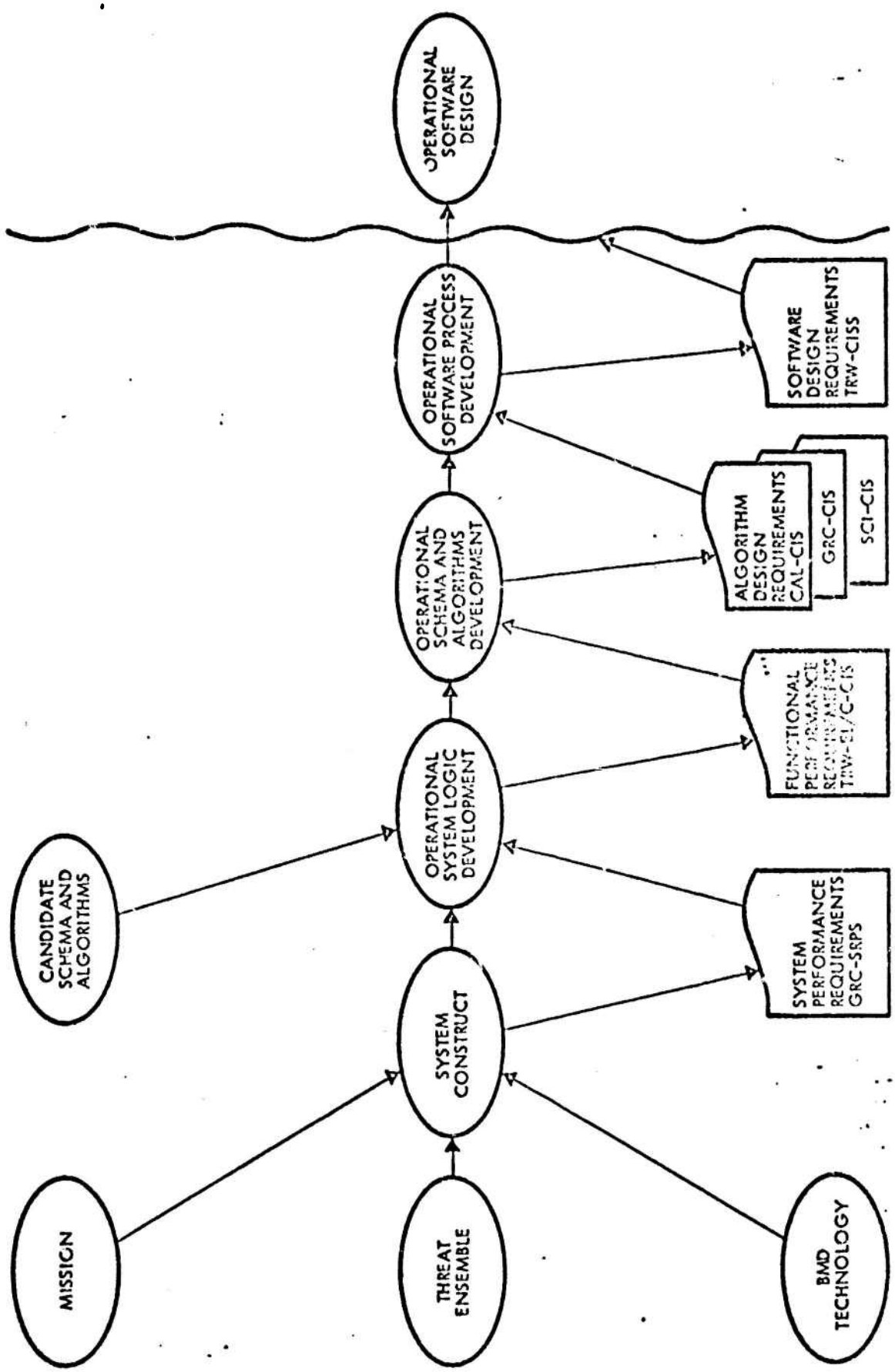


Figure 5-3 Terminal Defense Program Software Development Process

to the development process. The criticality of the operational schema and algorithms which must be available or developed in the process shown for TDP can determine whether the software development succeeds or fails. Thus, if an anticipated algorithm cannot be developed because of technological constraints, or whatever, a critical algorithm may result in program failure. This highlights the desirability of a feasibility study and of "Go-No Go" decision points in the software development process. This is especially true in software programs that fall into the Software or Research and Development categories defined by Weinwurm and discussed in Section 5.1.

Bemer [M.3] says the real problem in managing software is "As a relatively new profession (?) we are obsessed with reinvention, and forget that there is something known as management science." He proposes that we ask a series of questions that are not fully answered until feedback from the next question is received. These questions in order are:

- What should be produced?
- Should it be produced?
- Can it be produced?
- How should the producer be organized?
- How should the product be introduced?
- How should the product be improved and serviced?

Each of these questions can be further reduced into more specific questions. However, it is clear that answering these questions can lead to the definition of "Go-No Go" points in the system life cycle and to the determination of whether the software should be developed.

6.0 REFERENCES

This list of references is partitioned into three subsets corresponding to divisions laid down in the contents of this text. Due to the overlap in methodology and management techniques, these have been combined.

METHODOLOGY AND MANAGEMENT TECHNIQUES

M.1 Baker, F. T. and Mills, H. D., "Chief Programmer Teams," IBM Federal Systems Division, Gaithersburg, Maryland, February, 1973.

M.2 Baker, F. T., "Chief Programmer Team Management of Production Programming," IBM Syst. J. 11(1972) 56-73.

M.3 Bemer, R. W., "Manageable Software Engineering," in Software Engineering, Tou (ed.), 1973.

M.4 Boehm, B. W., "A Concept of Model Driven Software," Proc. TRW Symposium on Reliable, Cost Effective, Secure Software, Los Angeles, California, March, 1974.

M.5 Briggs, R. B., "A Mathematical Model for the Design of Information Management Systems," M. S. Thesis, University of Pittsburg, 1966.

M.6 Burger, R. T., "AUTASIM: A System for Computered Assembly of Simulation Models," Winter Simulation Conference, Sig Plan Notice, January 1974.

M.7 Computer Sciences Corporation, "A Users Guide to the Threads Management System," November, 1973.

M.8 Couger, J. Daniel and Knapp, Robert W., editors, Systems Analysis Techniques, John Wiley and Sons, New York, 1974.

M.9 Couger, J. Daniel, "Evolution of Business System Analysis Techniques," Computing Surveys, 5 (1973) 167-198.

M.10 Glans, T. B. et al, Management Systems, Holt, Rhinehart, and Winston, Inc., 1968.

M.11 Hartman, Matthes, and Proene, Management Information Systems Handbook, McGraw Hill, 1968.

M.12 Head, Robert V., "Automated System Analysis," Datamation, August 1971, p. 23.

M.13 IBM, "HIPO: Design Aid and Documentation Tool," IBM, SR20-9413-0, 1973.

M.14 IBM, "Study Organization Plan Documentation Techniques," IBM C20-8075 (1961), 1-26, and F20-8136 (1963), 2, 7-11.

METHODOLOGY AND MANAGEMENT TECHNIQUES (CONTINUED)

M.15 IBM, "The Time Automated Grid System (TAG): Sales and Systems Guide," IBM publication GY20-0358-1, 2nd Ed., May 1971.

M.16 Karp, R. M. and Miller, R. E., "Parallel Program Schemata," J. Comp. and Sys. Sci. 3 (1969) 147-195.

M.17 Langefors, B., "Information System Design Computations Using Generalized Matrix Algebra," BIT, 5(1965) 96-121.

M.18 Langefors, B., "Some Approaches to the Theory of Information Systems," BIT, 3(1963) 229-254.

M.19 Lynch, Hugh J., "ADS: A Technique in System Documentation," Database 1(1969) 6-18.

M.20 Nunnamaker, J. F., "A Methodology for the Design and Optimization of Information Processing Systems," AFIPS Proceedings, SJCC, 38(1971) 283-293.

M.21 Petri, C. A., "Kommunikation Mit Automation," Schrifton des Reinsch West Falischen Institute, Instrumentelle Math under Universtat Bonn Nr 2, 1962.

M.22 Rose, C. W., "A System Representation for General Purpose Digital Computer Systems," Jennings Computer Center Report No. 1113, Case Western Reserve University, Cleveland, Ohio, August, 1970.

M.23 Rose, C. W., "LOGOS and the Software Engineer," AFIPS Proceedings, FJCC, 41(1972) 311-323.

M.24 Royce, W. W., "Managing the Development of Large Software Systems Concepts and Techniques," TRW Software Series, August, 1970.

M.25 Rubin, M. L., Introduction To The System Life Cycle, Brandon/System Press, Princeton, NJ.

M.26 Teichroew, Daniel, "Problem Statement Language in MIS," in Systems Analysis Techniques, (ed. Couger and Knapp), John Wiley and Sons, New York, 1974.

M.27 Teichroew, Daniel and Carlson, David M., "A Model of the System Building Process," ISDOS Working Paper No. 64, Dept. of Industrial and Operations Engineering, University of Michigan, Ann Arbor, November 1972.

M.28 Teichroew, Daniel, Ernest A. Hershey III, and Michel J. Bastarache, "An Introduction to PSL/PSA," ISDOS Working Paper No. 86, Dept. of Industrial and Operations Engineering, University of Michigan, Ann Arbor, March 1974.

METHODOLOGY AND MANAGEMENT TECHNIQUES (CONTINUED)

M.29 Teichroew, D. and Sayani, H., "Automation of System Building," Datamation, (August 1971) 25-30.

M.30 TRW Systems Group, "Final Report for Engagement Logic and Control Studies," CDRL Item A004, October 1970.

M.31 TRW Systems Group, "Interpretation and Use of Engagement Logic," in "Engagement Logic and System Specification Validation," CDRL Item C005, September 1972.

M.32 Weinwurm, George F., "On the Economic Analysis of Computer Programming," in On the Management of Computer Programming, (ed. Weinwurm), Auerback Publishers, Inc., New York, 1971.

LANGUAGES

L.1 "ACM-IEEE Symposium on High Level Language Computer Architecture," (Nov. 7-8, 1973) 117-123.

L.2 Bauer, Charles R., Peluso, Anthony P. and Gomberg, David A., Basic PL/1 Programming, Addison-Wesley Publishing Co., Reading, Mass., 1968.

L.3 Bond, E. R. et al., "FORMAC - An Experimental FORmula MAnipulation Compiler," Proc. ACM 19th Nat'l Conf., 1964, K2.1-1 - K2.1-11.

L.4 Branquart, P., Lewi, Jr., Sintzoff, M. and Wodon, P. L., "The Composition of Semantics in ALGOL 68," CACM, 14 (Nov. 1971) 697-708.

L.5 Carleton, J. L., Lege, P. E., and Suarez, R. S., "A FORTRAN Extension to Facilitate Proposal Preparation," IEEE Trans. Elec. Comp., Vol. EC-13, No. 4 (Aug. 1964) 456-462.

L.6 Control Data, "Control Data Cyber 70 Computer Systems, A Simulation Process-Oriented Language (ASPOL) Reference Manual," Control Data Corp., Sunnyvale, CA, 1972.

L.7 Control Data, "Control Data 6400/6500/6600 Computer Systems ALGOL 60 Reference Manual," Control Data Corp., Palo Alto, CA (1967).

L.8 Control Data, "Control Data 6400/6500/6600 Computer Systems COBOL Reference Manual," Control Data Corp., Palo Alto, CA, 1969.

L.9 Control Data, "Control Data 6400/6500/6600 Computer Systems, JOVIAL General Information Manual," Control Data Corp., Palo Alto, CA, 1969.

L.10 Control Data, "Control Data 6400/6500/6600 Computer Systems B, SIMSCRIPT Reference Manual," Control Data Corp., Sunnyvale, CA, 1971.

L.11 Control Data, "Control Data 6400/6500/6600 Computer Systems SIMULA General Information Manual," Control Data Corp., Palo Alto, CA, 1970.

L.12 Control Data, "Cyber 70 Computer Systems FORTRAN Extended Version 4 Reference Manual," Control Data Corp., Sunnyvale, CA 1974.

L.13 Control Data, "PROSE, A General Purpose Higher Level Language Calculus Operations Manual," Cybernet Service, Control Data Corp., Minneapolis, Minnesota, 1974.

L.14 Control Data, "PROSE, A General Purpose Higher Level Language Procedure Manual," Cybernet Service, Control Data Corp., Minneapolis, Minnesota, 1974.

LANGUAGES (CONTINUED)

L.15 Corbato, F. J., "PL/1 As a Tool for System Programming," Datamation, (May 1969) 68-76.

L.16 Dahl, O. J., Myhrhaug, B., Nygaard, K., "Some Features of the SIMULA 67 Language,: Norwegian Computing Center, Oslo, September 1968.

L.17 Dahl, O. J., Myhrauz, B., Nygaard, K., SIMULA 67 Common Base Language, Norweigian Computing Center Report 52 (1968).

L.18 DesRoches, J. C., "Survey of Simulation Languages and Programs," The Mitre Corp., ESD-TR-71-227, July 1971.

L.19 Edelman, Howard, "A Short Guide to the Wonderful World of COBOL," Datamation, (Dec. 1969) 161-164.

L.20 Falkoff, A. D., and Iverson, K. E., "The Design of APL," IBM J. Res. Develop., (July 1973) 324-333.

L.21 Feingold, Robert S. and Chao, Yen W., "Statistical Instrumentation of ECSS Models," Symposium on the Simulation of Computer Systems II, (June 4-6, 1974) 147-161.

L.22 Heidorn, George E., "English As a Very High Level Language for Simulation Programming," IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

L.23 Hicks, Harry T., "ANSI COBOL," Datamation (Nov. 1970) 32-36.

L.24 Hicks, Harrt T., "A Communication Facility for COBOL," Datamation (Dec. 1969) 148-158.

L.25 Hurwitz, A., Citron, J. P., and Yeaton, J. B., "GRAF: Graphic Additions to FORTRAN," Proc. SJCC, 30(1967) 553-557.

L.26 IBM, "Computer System Simulator II (CSSII) General Information," IBM, White Plains, New York, 1970.

L.27 IBM, "Computer System Simulator/360 Program Description and Operations Manual," IBM, White Plains, New York.

L.28 IBM, "General Purpose Simulation System/360 User's Manual," IBM, White Plains, NY, January 1970.

L.29 Ichbiah, Jean D., "Extensibility in SIMULA 67," Compagnie Internationale pour l'Informatique, France, 84-86.

L.30 Intermetrics, "Final Report Vol II A Guide to the HAL Programming Language," Intermetrics Inc., Cambridge, Mass., June, 1971.

L.31 Iverson, K. E., A Programming Language, John Wiley and Sons, New York, 1962.

L.32 Kelley, R. A., "APLGOL, An Experimental Structured Programming Language," IBM J. Res. Develop., (Jan 1973) 69-73.

LANGUAGES (CONTINUED)

L.33 Kemeny, John G. and Kurtz, Thomas E., BASIC Programming, John Wiley and Sons, Inc., New York, 1968.

L.34 Kiviat, Philip J., "Digital Computer Simulation: Computer Programming Languages," RAND Corporation, RM-5883-PR, January 1969.

L.35 Kolsky, H. G., "Problem Formulation Using APL," IBM System J., No. 3, (1969) 204-219.

L.36 Kosy, D. W., "Experience With the Extendable Computer System Simulator," RAND Corp., R-560-NASA/PR, Dec. 1970.

L.37 Kosy, D. W., "Languages for Computer Systems Simulation," Symposium on the Simulation of Computer Systems II, Washington D.C., June 1974.

L.38 Nielsen, N. R., ECSS: An Extendable Computer System Simulator, The RAND Corp., RM-6132-NASA, February 1970.

L.39 MacDougall and MacAlpine, "Computer System Simulation with ASPOL," Proc. Symposium on the Simulation of Computer Systems, (1973) 93-103.

L.40 MacLaren, M. Donald, "Tasking in Standard PL/1," Sigplan Notices, 8, 9 (September 1973) 104-108.

L.41 Markowitz, H. M., Hausner, Band Karr, H. W., SIMSCRIPT - A Simulation Programming Language, Prentice-Hall, Inc., Englewood Cliffs, N. J.

L.42 Mathur, F. P., "A Brief Description and Comparison of Programming Languages FORTRAN, ALGOL, COBOL, PL/1, and LISP 1.5 From a Critical Standpoint," Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, September 15, 1972.

L.43 McCracken, Daniel D., A Guide to FORTRAN IV Programming, John Wiley and Sons, Inc., (August 1968).

L.44 McCracken, Daniel D., "Whither APL?," Datamation, (Sept. 15, 1970) 53-57.

L.45 Peterka, J. J. and Robbins, F. E., "SPCL - Simulation and Process Control Language Translator User's Manual," TM-HU-109/000/02, (July 15, 1973) System Dev. Corp., Huntsville, Alabama.

L.46 Pomeroy, J. W., "A Guide to Programming Tools and Techniques," IBM System J., No. 3, (1972) 234-254.

L.47 Prenner, Charles J., "The Control Structure Facilities of ECL," Center for Research in Computer Technology, Harvard University.

L.48 Remy, Eldon H., "Learning to Use PL/1," Datamation, (July 1970) 47-51.

LANGUAGES (CONTINUED)

L.49 Rosen, Saul, "Programming Systems and Languages 1965-1975," CACM 15 (July 1972) 591-600.

L.50 Rosen, Saul, Ed., Programming Systems and Languages, McGraw-Hill Book Co., 1967.

L.51 Roth, Paul F., "The BOSS Simulator--An Introduction," Burroughs Corp., Paoli, Pennsylvania.

L.52 Sammet, Jean E., Programming Languages: History and Fundamentals, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1969.

L.53 Sammet, Jean E., "Programming Languages: History and Future," CACM, 15 (July, 1972) 601-610.

L.54 Sammet, Jean E., "Roster of Programming Languages," Computers and Automation, 20 (1971).

L.55 Sammet, J. E., and Bone, E., "Introduction to FORMAC," IEEE Trans. Elec. Comp., Vol. EC-13, No. 4 (August, 1964) 386-394.

L.56 Seaman, P. H. and Soucy, R. C., "Simulating Operating Systems," IBM Syst. J., No. 4, (1969) 264-279.

L.57 Schaefer, Marvin, "DBL: A Language for Converting Data Bases," Datamation, (June, 1970) 123-130.

L.58 Schutte, Lawrence J., "A Report on the Value of Some Advanced High Level Language Operators on Current Sequential Computers," ACM-IEEE Symposium on High-Level-Language Computer Architecture, (Nov. 1973), University of Maryland, College Park, Maryland.

L.59 Sperry Rand, "UNIVAC 1108 Exec II COBOL Programmers Reference," UP-4048, Sperry Rand Corp., 1969.

L.60 Stygas, Paul, "Independent Research and Development Project Information Processing Modules System for User Data Structuring--SUDS--Prototype," TRW Systems Group, McLean, VA, March 1974.

L.61 Syn, W. M., and Linebarger, R. N., "DSL/90--A Digital Simulation Program for Continuous System Modeling," Proc. SJCC, 28(1966) 165-187.

L.62 Texas Instruments, "Process Design System User's Manual," Texas Instruments Inc., Huntsville, Alabama, March 1974.

L.63 Thames, Jac., and Robinson, Michael, PROSE--A Very High Level General Purpose Language, PROSE Inc., Los Angeles, CA, May 23, 1974.

L.64 TRW Systems Group, "PCP--Computer Program Requirement Document for Process Construction Program," TRW No. 10780001B, TRW, Los Angeles, CA, May 25, 1973.

LANGUAGES (CONTINUED)

L.65 TRW Systems Group, "SALSIM Systems Analysis Language for Simulation Logic Control Chain Functional Operator Manual," TRW Systems Group, Redondo Beach, CA, April 1972.

L.66 Wegbreit, Ben, "An Overview of the ECL Programming System," Center for Research in Computing Technology, Harvard University.

L.67 Wegbreit, Ben, "The Treatment of Data Types in PL/1," CACM 17 (May 1974) 251-262.

L.68 Wulf, W. A., Russell, D. B. Habermann, A. N., "BLISS: A Language for Systems Programming," CACM, 14 (1971) 780-790.

TESTING

T.1 Brown, J. R. and R. H. Hoffman, "Evaluating the Effectiveness of Software Verification--Practical Experience with an Automated Tool," Proceedings of FJCC, ACM, 41 (1972) Part I, 181-190.

T.2 Brown, J. R. et. al., "Automated Software Quality Assurance" in Program Test Methods (ed. Hetzel) Prentice-Hall, Englewood Cliffs, NJ 1972.

T.3 Carey, Levi J., "IEEE Symposium on Software Reliability," Datamation, October, 1973, p 119.

T.4 CODASYL Committee,"An Information Algebra - Phase I Report," CACM 5 (1962), 190-204.

T.5 Dijkstra, E. W., "Concern for Correctness as a Guiding Principle for Program Composition," in The Fourth Generation, Infotech, 1971.

T.6 Dijkstra, E. W., "Notes on Structured Programming," in Structured Programming, by Dahl, Dijkstra, Hoare, Academic Press, New York, 1972.

T.7 Elspas, Bernard et al, "An Assessment of Techniques for Proving Program Correctness," Computing Surveys 4 (1972) 97-147.

T.8 Floyd, Robert W., "Assigning Meanings to Programs," Proceedings of a Symposium in Applied Math, AMS, 19 (1967) 19-32.

T.9 Gibson, C. G. and Railing, L. R., "Verification Guidelines," TRW-SS-71-04, August, 1971.

T.10 Grindley, C. B. B., "SYSTEMATICS - A Non-programming Language for Designing and Specifying Commercial Systems for Computers", Computer Journal, (1966) 124-128.

T.11 Haney, Frederick M., "Module Connection Analysis - A Tool for Scheduling Software Debugging Activities," Proceedings of FJCC, ACM, 41 (1972), Part I, 173-179.

T.12 Hetzel, William C. (Ed.), Program Test Methods, Prentice-Hall, Englewood Cliffs, NJ, 1973.

T.13 King, J. C., "A Program Verifier," Ph.D. Thesis, Department of Computer Sciences, Carnegie-Mellon, 1969.

T.14 Krause, K. W., Smith, R. W., and Goodwin, M. A., "Optimal Software Test Planning Through Automated Network Analysis," in Record: 1973 IEEE Symposium on Computer Software Reliability, New York City, 1973.

T.15 Langefors, Borje, Theoretical Analysis of Information Systems, Auerbach Publishers Inc., Philadelphia, 1973.

TESTING (CONTINUED)

T.16 Linden, T.A., "A Summary of Progress Toward Proving Program Correctness," Proceeding of FJCC, ACM, 41 (1972) Part I, 201-211.

T.17 London, Ralph L., "The Current State of Proving Programs Correct," Proceedings of ACM Annual Conference (1972) 39-46.

T.18 Manna, Zohar, "Properties of Programs and First Order Predicate Calculus," JACM, 16 (1969) 244-255.

T.19 Manna, Zohar, "The Correctness of Programs," J. Comp. and Syst. Sci., 3 (1969) 119-127.

T.20 Martin, David F., "Formal Languages and Their Related Automata," in Computer Science, (ed. Cardenas, Press, Marin) Wiley-Interscience, New York 1972.

T.21 Maurer, W. D., "A Semantic Extension of BNF," International Journal of Comp. Math., 1972.

T.22 Mills, Harlen, "Top Down Programming in Large Systems," in Debugging Techniques in Large Systems, (ed. Randall Rustin), Prentice Hall, Englewood Cliffs, NJ, 1971.

T.23 Minneman, Milton J., (Chairman), Record: 1973 IEEE Symposium on Computer Software Reliability, New York City, 1973.

T.24 Naur, Peter, "Proof of Algorithms by General Snapshots," BIT, 6 (1966) 810-816.

T.25 Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," CACM, 15 (1972) 1053-1058.

T.26 Rustin, Randall (ed.), Debugging Techniques in Large Systems, Courant Computer Symposium, Prentice Hall, Englewood Cliffs, NJ, 1971.

T.27 Rutledge, J. D., "On Ianov's Program Schemata," JACM, 11 (1964) 1-9.

T.28 Smith, J. Meredith, "Proof and Validation of Program Correctness," The Comp. Journal 15 (1972) 130-131.

T.29 Wolverton, R. W. and Schick, G. J., "Assessment of Software Reliability," TRW-SS-72-04, September, 1972.

22944-6921-011

SOFTWARE PERFORMANCE REQUIREMENTS - SOFTWARE REQUIREMENTS ENGINEERING METHODOLOGY

CDRL G009

AUGUST 15, 1974

Sponsored By
BALLISTIC MISSILE DEFENSE
ADVANCED TECHNOLOGY CENTER

DAHC60-71-C-0049

TRW
SYSTEMS GROUP
Huntsville, Alabama

22944-6921-011

SOFTWARE PERFORMANCE REQUIREMENTS - SOFTWARE REQUIREMENTS ENGINEERING METHODOLOGY

CDRL G009

AUGUST 15, 1974

Sponsored By
Ballistic Missile Defense
Advanced Technology Center

DAHC60-71-C-0049

TRW
SYSTEMS GROUP
Huntsville, Alabama

SOFTWARE PERFORMANCE REQUIREMENTS - SOFTWARE REQUIREMENTS ENGINEERING METHODOLOGY

CDRL G009

AUGUST 15, 1974

Principal Author

R. J. Smith

Principal Contributors

M. Alford
T. Bell
M. Dyer
M. Hansing
F. Herring
J. Richardson

Approved By

for Mack Alford
T. W. Kampe, Manager
Software Requirements
Engineering Program

James E. Long
J. E. Long, Manager
Huntsville Army Support Facility

Sponsored By
Ballistic Missile Defense
Advanced Technology Center
DAHC60-71-C-0049

TRW
SYSTEMS GROUP
Huntsville, Alabama

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

1.0 PURPOSE AND SUMMARY

This document describes planned research to be carried out by TRW as part of its Software Requirements Engineering Program. The research described herein is directed at developing a methodology, and the supporting computer-aided tools, required to produce computer-independent software requirements specifications in a manner and form which will improve the overall development process for real-time software. The focus of the research will be on BMD software, however, it is fully anticipated that the results will be applicable to large-scale software development in general, and therefore, maintenance of a reasonable degree of generality will be a goal.

The problem statement and description of the anticipated research activities and products contained herein provide BMDATC and TRW with a concise statement of the research mission and should serve as a basis for review and mutual understanding of the research. Because research directions and objectives can change as intermediate results are obtained and assessed, this plan will be periodically reassessed in order to continually fulfill it's objective.

The relationship of this research to the overall BMDATC mission and the problems with existing software development efforts are discussed in Section 2.

The planned research activities, approaches and objectives are presented in Section 3. The described approach emphasizes:

- A continuing research effort with incremental development of methodology components and high visibility to the customer of intermediate results.
- Empirical assessment of methodology concepts through the use of test cases and experiments.
- A continuing top-level assessment of software problems and their relationship to requirements engineering in order to assure potentially high-payoff areas are being addressed.
- An active information interchange with contractors involved in other areas of the overall BMDATC program.
- Preliminary design of all proposed computer-aided tools with implementation of a selected subset based on a cost/benefit analysis.

2.0 INTRODUCTION

2.1 THE ROLE OF SOFTWARE REQUIREMENTS ENGINEERING RESEARCH IN THE OVERALL BMDATC RESEARCH MISSION

The Software Requirements Engineering Program (SREP) described herein forms an integral part of the BMDATC Software Development and Evaluation Technology Program. The objective of the overall BMDATC program is to further advance the state-of-the-art in the specification, design, implementation and test of real-time BMD software. The need for such a program stems from the recognition that software increasingly consumes a larger percentage of total weapon system costs and has become more frequently the critical path item in a system development schedule. To achieve its objective, BMDATC has formulated a composite software research program which includes, besides the TRW program of research in software requirements engineering, research activities in process engineering and in software analysis and testing. The objectives of the composite program are to develop the technology for:

- Generating valid, unambiguous, and traceable software requirements.
- Automating design and implementation of structured and reliable software processes.
- Validation of specification compliance and verification of software effectiveness.
- Technique evaluation; technology transfer; standardization; management, cost and scheduling methodology; and performance analysis.

The overall BMDATC program encompasses the entire spectrum of software development activities. The activities, under the general BMDATC approach developed to date, can be partitioned into appropriate subsets as described below:

a) <u>System Construction Definition</u>

The top-level system engineering activity provides the design of the system construct, derives top-level system performance requirements, identifies the subsystems, and allocates the system requirements to the various subsystems. It is at this point that the system level performance requirements

on the software are established. This activity produces the Systems Requirements and Performance Specification (SRPS) which provides the initiating requirements and information for the following activities.

b) Software Requirements Engineering (First Phase)

The next level activity with respect to software development expands upon the system level requirements. The operational logic and defense tactics are incorporated into the performance requirements to develop the requisite logic to be provided by the software. The system level functions are partitioned into a set of required algorithms or computational procedures, and the system level performance requirements are allocated onto those algorithms. The end product of this activity is a Computer-Independent Specification of the Software Functional Requirements (CIS-SFR).

c) Algorithm Selection

The computational procedures are selected and derived to meet the requirements imposed on each algorithm as specified in the preceding activity. Theoretical techniques and known candidate algorithms are analyzed using theoretical and empirical phenomenological data to determine performance. Once an algorithm is chosen, it is "tuned" for the specific application by determining the value of controlling parameters, decision criteria, etc. The end product of this activity is a Computer-Independent Specification (CIS) for each algorithm.

d) Software Requirements Engineering (Second Phase)

The algorithm designs are integrated into the previously specified logic in the CIS-SFR, data transfers between algorithms are provided, and control flow from one function to the next is derived wherever this flow is significant with respect to the specified system behavior. All decision thresholds, operational parameters, etc., are established, and total system (i.e., software subsystem) coherency is established. The end product of this activity is a Computer-Independent Software Specification (CISS), validated by a CISS Simulator.

e) Process Design

The software requirements specified in the CISS are analyzed with respect to the constraints and characteristics of the data processor to be used, and a top-level software structure is derived which will permit optimal execution on the chosen computer. The global file structure and the control structure are defined and the data and control flows are developed which will maximize the efficiency of the software/hardware

while assuring that the specified performance requirements are met. These performance requirements are allocated to the individual top-level software modules defined by the chosen software structure. The end product of this activity is a Computer-Dependent Process Design Specification (CDPDS).

f) Software (Program) Design

The top-level software modules defined in process design are further partitioned into smaller units (e.g., routines) for efficiency of execution and to facilitate the management of their implementation. The specific requirements on each lower-level module are derived from the requirements specified for the parent top-level modules. The end product of this activity is a detailed coding specification.

g) Software Implementation

The individual modules are coded, tested, and integrated into a complete structure. Once completed, the entire assemblage is tested against the parent software requirements (CISS).

h) Software Validation

The completed software is tested against the original subsystem (CISS) and system (SRPS) requirements to verify that the end product does in fact meet the required objectives.

To support this overall, composite program, the TRW research (SREP) addresses the development of a systematic, unified, computer-aided approach to the development of software requirements specifications, and is concerned with all the activities from the SRPS to the CISS.

2.2 PROBLEM STATEMENT

The specific software problem area attacked by the SREP research is software requirements engineering, which involves all aspects of the generation, validation, and communication of software requirements. The need for improvements in software requirements engineering, however, stems mainly from problems and unsatisfactory conditions that exist in the overall software development process. Examination of software development problems and their symptoms has shown a connection with, and a need for improvement in, requirements engineering. These problems and their relationship to requirements engineering are discussed in the following paragraphs.

2.2.1 Problems in Software Development

It is generally recognized that the development of data processing and information systems has too often resulted in cost overruns, schedule slippages, or failure to produce a system which satisfies the original objectives. As Boehm stated (Reference 1), "Software (as opposed to computer hardware, displays, architecture, etc.) was 'the tall pole in the tent'--the major source of difficult future problems and operational performance penalties." The annual expenditures for the Air Force and NASA show (Reference 1) twice as much money is spent on software as hardware. This indicates the tremendous impact that problems in developing software can have on the budgets of the services or on the cost of any large software system that is being developed.

A number of specific problems, their symptoms and suspected root causes have been identified in various studies researched under this project. Among these studies are the McGonagle (Reference 2) study for the Air Force investigation, studies by Logicon (Reference 3) and studies for NASA. Foremost among the problems in software development is the generally undisciplined approach to the development of software. This approach makes the success of a project a function of the development personnel involved, or of the customer's flexibility and tolerance level.

This lack of discipline is particularly present in the requirements development activities. Too often, the programming effort has begun without specific, detailed, non-ambiguous requirements. As a result, there have been many changes or modifications to the system requiring large rewrites, and creating problems in debugging due to non-testable system objectives.

Even when software requirements have been stated, there is no efficient and reliable way for a customer to determine whether the specification is complete, consistent and responsive to his needs. Also, there is no framework which can be used to systematically compare the specifications produced by more than one organization. Specifications are generally manually generated, frequently imply far too much design, and often implicitly state a requirement rather than giving explicitly testable requirements. There generally is not a way to trace the requirements specified at one level of the data processing systems project development to a higher level requirement, or to the system objectives. Traceability is normally provided by manual means, thus again becoming a function of how competent the individual is, rather than reducing this reliance upon people by an automated technique.

Finally, consider the results given by McGonagle (Reference 2) in his study for the CCIP-85 effort. He discovered that nineteen percent of the software errors resulted from unexpected side effects from changes. Twenty percent of the errors were due to logical flaws in the design. Inconsistencies between design and implementation resulted in twenty-two percent of the errors. In other words, 61% of the total errors were basically due to design flaws, specification flaws or to effects due to changes. Clearly, a rigorous approach that provides a means for readily accepting changes, and for producing unambiguous, consistent, complete requirements should reduce the cost and time required to develop software.

2.2.2 Basic Problems Involving Software Requirements

As discussed above, the ultimate symptoms of problems or shortcomings in specifying software requirements are excessive cost, low reliability, and missed deadlines in the resultant software. These ultimate symptoms can usually be identified as arising from second level effects such as having to reprogram major sections of a system, having the wrong functions performed by the system, or having to modify interface routines added to the software late in its development. These effects, however, are not the most appropriate level to work on resolving the problems, since they are actually symptoms of basic problems at yet a third level. These basic problems involving software requirements are the most critical (and least addressed) ones that cause the

ultimate symptoms noted above. The list of these third-level problems given below indicates the need for the research undertaken in this project.

- Failure to do traceable, top-level requirement analysis may lead to incomplete and/or inconsistent requirements.
  - Initial Weapon System requirements are incomplete with respect to the Data Processing Subsystem because not all the characteristics of the other subsystems are known. This makes totally top-level requirements analysis difficult or impossible.
  - Inconsistent requirements between the Weapon System and Data Processing Subsystem occur as a result of simple mistakes.
  - Internally inconsistent requirements within the Data Processing Subsystem result from a lack of cross-checking between requirement statements.
  - Incomplete requirements result when important things are left out because:
    a) Analysts simply forgot some important things.
    b) Analysts (individually) thought that some requirements were in another analyst's requirements areas.
    c) Analysts thought in terms of an unarticulated but assumed design and omitted the requirements because the postulated design did not require stating them.
    d) Analysts concentrated on the requirements they felt most important and assessed importance incorrectly.
- Incorrect interpretation of requirements (or inconsistent interpretation between several people) may exist when requirements analysts are interpreting Weapon System requirements (to get to a CISS) or Software Designers are reading the specification.
  - Statements are ambiguous and are interpreted differently due to backgrounds or different objectives.
  - Semantics of "key words" are different in different disciplines and lead to different interpretations of requirements' meanings.
  - The boundaries of subsystems within the Weapon System or the Data Processing Subsystem are not clearly defined so several subsystem developers cover the interface or, alternatively, everyone omits it.

- Synthesis of components may lead to bad system behavior.
  - Although the individual information subsystem components (e.g., a filter) perform adequately (and to spec), they do not perform well together after synthesis due to feedback problems, bad interfaces, or detailed analytical problems (e.g., parameter values are wrong).
  - Interactions between the Data Processing Subsystem and the other subsystems are inappropirate due to unanticipated interface problems or feedback problems.
  - Data Processing Subsystem requirements must be changed during software development due to changes in the threat or changes in the rest of the Weapon System (such as the realization that other Weapon System components are not realizable). Such changes may be made inappropriately when requirement traceability is incomplete.
- Assumptions about Data Processing Subsystem technology are often inappropriate.
  - Assumptions about performance capabilities (either timing or analytical) of Data Processing Subsystem components (either hardware or software) are optimistic and necessitate changes in requirements of other components or of the Weapon System as a whole.
  - Technological problems turn out to be more difficult than expected to solve, and the Data Processing Subsystem functional design requirements must be revised to pursue an alternative approach.
  - Implementation decisions (e.g., computer hardware, "over-specified" software) are made too early and are therefore inappropriate.
  - The loading effects of system interactions are underestimated on some system components, so requirements are not set appropriately.
- Management visibility is often inadequate.
  - Management cannot discern whether a requirement analysis job is being performed realistically and on time since meaningful milestones and milestone products have not been defined.
  - Requirements are not specific enough to enable early testing to indicate when, during implementation, a development effort begins to go wrong.

To attack each of these problems individually would be inefficient and would probably result in an uncoordinate set of inconsistent or overlapping solutions which would be of little improvement. Therefore, the core of the research problem is to develop an integrated methodology which addresses the above problems. The desired methodology, in order to solve these problems, should provide:

- A structured approach to developing and validating software requirements
- Integrated use of computer-aided tools to assist in the requirements effort
- A structured medium, or language, for specifying requirements to the process designer.

3.0 SCOPE OF RESEARCH ANALYSIS MISSION

3.1 RESEARCH OVERVIEW

The future research program is based on the project results of the last two years. The major lessons learned, relative to software requirements methodology are:

- There are many types of software developments. A deployable system (e.g., Site Defense) which is developed top-down is different from TDP which serves, in part, as an algorithm test bed.
- This methodology must thus consist of a family of tools, to be used in varying orders and combinations, depending on the type of software development at hand.
- There are many tools of potential value; the ones to be actually implemented as a part of this project must be selected by cost/benefit analysis.

The cost and benefit values must be determined during the research program. As a consequence, the research plan and approach must be selected to exploit developments, while not getting locked into unfruitful lines of investigation.

Because of the uncertainties associated with scheduling breakthroughs, it is unwise to formulate the research program as one which grinds away to produce a single finished product. Rather, a series of operational capabilities is envisioned, each of succeeding capability. In addition, there is a set of intermediate milestones. These two measures offer good visibility into the research effort, and offer opportunities for technical redirection.

In the research plan presented herein, not all tasks are equally well-defined. Emphasis has been placed on immediate activities. Longer range tasks will be planned in greater detail as the research progresses.

A specific example of this deferral of decisions is the selection of which tools to implement. This decision will be made after all tools have been defined, tests proposed, and a cost/benefit analysis made.

3.1.1 Research Objectives

This project has the principal objective of reducing the cost and risk of producing large and complex software packages through improved methodology for software requirements.

The objective is not necessarily to reduce cost and risk for producing requirements; the objective is to reduce overall software life-cycle cost and risk. Examination of the problem statement (Section 2.2) leads to a series of objectives for the requirements process itself, viz.:

- Reduction in ambiguity and contradiction in software specifications
- Reduction in time to produce specifications, and to respond to system changes
- Improvement in software testability
- Improvement in validation of software specifications through analysis and simulation
- Minimizing the amount of manual work required to produce all of the specification related products, i.e., interface documentation, sequencing logic, simulators
- Improvement in configuration control
- Improvement in management visibility for both specification production and for the resultant software design and implementation.

3.1.2 Major Study Areas

There are four study areas:

- Problem definition and specific objective selection (Section 3.2.1)
- Methodology research (Section 3.2.2)
- Computer-aided tools development (Section 3.2.3)
- Methodology experimentation and evaluation (Section 3.2.4)

Problem definition and specific objective selection is devoted to analyzing research results, problem analysis, and the evolving cost/benefit system to firm up project objectives.

Methodology research is concerned with defining the overall problem, the inputs (SRPS) and outputs (CISS) and with defining the methodology to go from the inputs to the outputs.

Computer-aided tools development has two functions. The first is to produce a preliminary design for the full set of automated tools to support the methodology. The second is to implement selected tools; only a subset of the full tools will be developed. The precise subset will be selected by a cost/benefit analysis.

Methodology experimentation has two functions. The first, and most formal, is to define and conduct experiments on the methodology to verify performance. The second function is to conduct empirical studies in requirements and methodology to provide data upon which the methodology is to be based.

3.1.3 Milestones and Decision Points

Figure 3-1 shows the overall schedule for the project. Major milestones consist of the following:

- Research Plan
- Methodology Capabilities Defined
- Language Requirements Defined
- Automation Requirements Defined
- Methodology Research Completed
- Experiment Plan Developed
- Language Designed
- Tools Designed
- Tools Implemented and Development Test Completed
- Experiment Completed and Evaluated
- Final Documentation and Dissemination

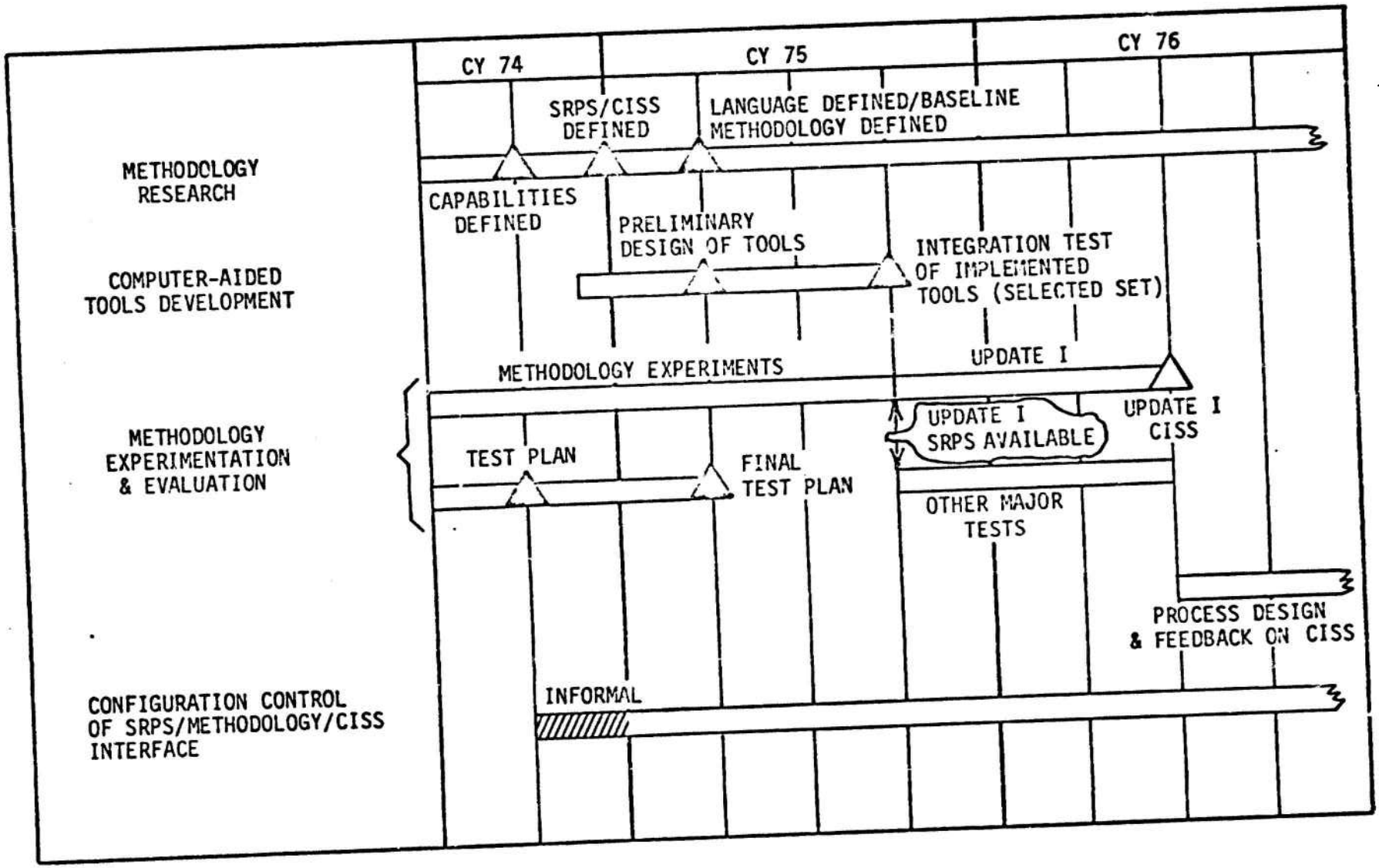


Figure 3-1 Top-Level Schedule

3.1.4 Dissemination Plan

Research information is not useful unless it gets to people who can use it. To assure that the information does get to the people who can use it, a dissemination plan is proposed.

The first element in the plan is the methodology notebook. Reports produced on the methodology, and the research which goes into its development, will become sections in a notebook. This will facilitate revisions, and offers a flexible approach to basic documentation.

The second element in the plan is to present papers at technical conferences, and submit papers for publication. This will expand greatly the number of people exposed to the methodology.

The third element is to include methodology material in short courses given by TRW staff members.

3.2 RESEARCH ANALYSIS TASK DESCRIPTIONS

3.2.1 Problem Definition and Specific Objectives Selection

This task is concerned with insuring that the research program objectives and activities are at all times consistent with the best understanding of the overall problem. A primary responsibility of this activity will be to evolve, and maintain current, a detailed, comprehensive statement of the problem at which this research effort is directed. Such a current, evolving problem statement is essential to insure that a mutual understanding of the research exists not only between BMDATC and TRW, but between the various elements within the project. The approach will be to identify the unsatisfactory conditions and outcomes presently associated with large, complex software development efforts (high cost and large risk being two such high level conditions). As a goal these problem descriptors (symptoms) will be further developed in an attempt to define quantifiable measures against which the methodology can be developed, and eventually experimental methodology evaluation can be conducted. The root causes of these unacceptable conditions (symptoms) must be further explored with the objective of identifying those problems which can potentially be ameliorated by improvements in software requirements development, validation, or their communication to the process designer. The further definition of the connection between the causes of software problems and the software requirements engineering activities is basic to insuring proper direction of the methodology development.

The problem statement presented in Section 2.2 has been developed from the analyses of existing methods reported in Reference 4. Under this task it will be further developed to more completely identify the problem descriptors against which the methodology will be directed. As the methodology and tools are developed, the incremental results and research directions indicated will be reviewed against the problem statement to determine:

- Have the root causes been addressed?
- Have some problems been overlooked or underestimated in importance?
- Have the new methods and proposed tools introduced new sources of problems in themselves?

- Is the problem statement sufficient to assess the methodology; i.e., can evaluation data of any form be accumulated which can be evaluated against the problem statement?

- Can the problem statement, problem descriptors and causes be better stated, quantified or more precisely defined?

While no guarantee can be made that all the software problem causes have been found and precisely defined, by keeping the problem statement alive through constant review and challenge, a high confidence can be established that the research efforts are being properly channeled. Key to this objective is the maintenance of a high degree of visibility to BMDATC of the evolving problem statement, and frequent technical interchanges across the contractor interfaces (TRW/TI and TRW/GRC).

3.2.2 Methodology Research

This task is the key element of the research program and encompasses the research activities directed at developing a sound engineering approach to the derivation and specification of software requirements. Examination of present software development techniques (Reference 4) and research to date on this program has revealed among other problem sources:

- Lack of a structured approach to developing software requirements

- Non-integrated use of computer-aided tools to assist in the requirements development

- Lack of a structured medium, or language, for specifying requirements to the process designer

Research under this task will be directed at these shortcomings. The basic approach to be followed in developing the methodology will be an incremental development which, in four major milestones:

- Defines methodology capabilities

- Identifies language requirements

- Defines automation requirements

- Synthesizes a baseline methodology

Theoretical analyses will be an important source for many of the methodology concepts, however, the approach will be empirically based in that as concepts are proposed, a series of "experiments" and demonstrations (see Section 3.2.4) are planned which will test the concepts on real-world type problems. The experiments will grow in complexity as the methodology evolves to provide a comprehensive testing of the overall approach. This approach is necessary to ensure that the methodology is "do-able," to provide insight into subtler problems, to uncover new problems which may be created by the methodology, and to provide hands-on experience for personnel who will utilize the methodology during the evaluation experiment planned for the complete baseline methodology.

The ultimate description of a methodology to go from a system description to a real-time software requirements specification would be a description of the steps from the one to the other, and a schedule of the outputs of each step together with a description of the way in which they are related. Whereas such a document is necessary for any particular project (in fact it constitutes the management plan for the software requirements phase), there are sufficient differences between projects (i.e., their goals, the information which is given and that which is yet to be determined, the size of the project, the total cost, the schedule) that a universal project plan cannot be written. For example, consider a project in which all of the software specifications must be derived from scratch; the manner in which this is done bears little resemblance to a project in which significant modifications to an already existing set of software are to be made.

The approach taken in this research is to take a project-independent view of the software requirements specification process by identifying an initial information set called SRPS, which is the system level description of the problem; to identify a Computer-Independent Software Specification, which is the necessary and sufficient set of processing requirements information to perform a computer-dependent process design; and to identify the discrete number of information sets between them, as well as the set of tasks and tools necessary to obtain those products. The differences in the project types center around the amount of information available at the start of the project, the set of information which is fixed (e.g., description of certain processing

steps), and the way in which certain sequences of steps are iterated in order to achieve consistency of the information sets.

The methodology, then, will consist of a description of the information sets; a description of the tasks which are necessary to proceed from one information set to another, including the purpose, information required, output, skills utilized within the task, and tools necessary (if any); and a description of the interactions of the tasks necessary to achieve the desired result. For example, the identification of the values of accuracy of the tracking computations might require the building of a simulator, which would be a task in itself; the interactions of these two tasks would have to be identified.

The following paragraphs describe the major activities within this task.

3.2.2.1 Definition of Methodology Capabilities

From analysis of the problem statement, solution attributes and the results of a cost-benefit analysis, the implementable objectives will be defined. The capabilities statement will define the overall capabilities the methodology and support tools should possess to achieve a cost-effective solution or reduction of the problem. Wherever feasible, quantitative criteria will be established against which the final products can be assessed.

Candidate capabilities to be possessed by the methodology for developing software requirements derived from research to date include:

- A high degree of automation is desirable to reduce human errors and relieve analysts of detailed, repetitive type work. In addition, automation will reduce the time and effort to derive requirements and systematize the methods used in the derivation.
- Techniques for controlling various configurations of requirements and software were designated as a necessary part of the methodology.
- The methodology must produce requirements which can be validated for accuracy, completeness, and consistency.
- The designs and requirements at each level in the development of a set of requirements and the subsequent software must be traceable back to the next higher level.

Specific capabilities that will be examined in developing specifications at all levels include:

- A multi-level user-oriented language which is structured for readability.
- The capability to check the logical consistency and completeness of requirements at each level of development through automated static testing, through an examination of input/output compatibilities, and through identification of conflicting, redundant, or unused processing steps.
- Traceability by automatically providing and maintaining the transformation mechanisms to correlate the requirements in the structure at one level of specification to the structure at the next level.
- Synthesis of the requirements, stated in terms of input, processing, output, into an algorithm sequencing logic (ASL).
- Automated simulator construction to the highest degree possible.
- A translator that produces both the requirements and the simulator to ensure that the simulator accurately represents the requirements being validated.
- Automated development of flowcharts and structured text for the requirements and simulators used to validate the requirements.
- The capability to generate open-loop test cases to examine the response of a part of the system or a single algorithm to stimuli.
- The automated insertion of data collection and reduction tools into simulators constructed to develop requirements.
- Data base management capabilities that generate and optimize data bases, automatically initialize the data base, and ensure data transfer compatibility.
- Features that aid in the preparation of test plans and results.

Capabilities were identified which support management visibility into the software requirements development and ensure that configuration management could be achieved. These capabilities included:

- Maintenance of a catalog of requirements that could be traced from the highest level of development to the real-time software with automated updating a principal feature.
- Automated reporting on costs, both actual and projected, relative to the allocated budgets for elements of the program.

- Automated reporting on status of each element being developed including schedule completion date, percentage completed, expected completion date, and critical items or paths.

- Security features that control access to different versions of the requirements and the software so that only authorized changes can be made to the software, requirements, or design.

3.2.2.2 Definition of Language Requirements

The detailed requirements on the software requirements language will be defined to provide a sound basis for subsequent design and development efforts on the language and language processors. The language should support:

- Automation of the generation of simulators

- Automation of specification documentation

- Maximization of information transfer without implying or overspecifying a structure (either logic or data base)

- Analysis of requirements statements for consistency and completeness.

Table 3.1 contains a preliminary comparison of "software requirements language" attributes with other languages, and Table 3.2 contains language requirement elements to be examined.

Language requirements will be defined, with particular emphasis on ensuring a compatible interface with the TI Process Design System, to support the BMDATC program goal of developing a unified, software development methodology from system specification through completion of process design. Language requirements will be strongly influenced by the CISS definition activities described in Section 3.2.2.3.

3.2.2.3 Definition of the Software Requirements Engineering/Process Design Information Interface (CISS)

The necessary and sufficient processing requirements information required by the process designer to most efficiently carry out a software design will be determined. The form, content and level of detail, of the Computer-Independent Software Specification (CISS) will be defined. The goal is to develop a structured communication medium which will be readable, unambiguous, preserve the computer-independent concept, not usurp design

Table 3.1 Preliminary Language Comparisons

ATTRIBUTES	PROGRAMMING LANGUAGE	SIMULATION LANGUAGE	PROCESS CONSTN. LANGUAGE	PROCESS DESIGN LANGUAGE	SOFTWARE RQMTS LANGUAGE
COMPUTATIONS	YES	YES	YES	YES	YES
LOGICAL OPERATIONS	YES	YES	YES	YES	YES
EXECUTION TIME RQMTS.	NO	MODELED	YES	YES	NO
DATA STRUCTURE DEFINITION	NO	NO	YES	YES	NO
DP ARCHITECTURAL RQMTS.	NO	NO	PARTIAL	YES	NO
TOP-DOWN DATA STRUCTURES	NO	NO	NO	YES	NO
OPERATING SYSTEM DESIGN	SPECIAL	SPECIAL	NO	YES	NO
ASSERTION SYNTAX	NO	NO	NO	YES	NO
MEMORY BUDGET RQMTS.	NO	NO	NO	YES	NO
BUILT-IN SIMULATION	NO	NO	NO	YES	YES
BUILT FOR AUTO. DOCUMENT'N	PARTIAL	NO	NO	YES	YES
BUILT-IN DEBUG CAPABILITY	PARTIAL	PARTIAL	NO	YES	YES
TIME RESPONSE RQMTS.	NO	NO	NO	OPTIONAL	YES
ACCURACY RQMTS.	NO	NO	NO	OPTIONAL	YES
BUILT-IN TRACEABILITY	NO	NO	NO	OPTIONAL	YES
BUILT-IN TEST CAPABILITIES	NO	NO	NO	OPTIONAL	OPTIONAL
LIBRARY MANAGEMENT	NO	NO	PARTIAL	YES	YES
BUILT-IN CONFIG. CONTROL	NO	NO	PARTIAL	YES	YES
FORCED MODULARITY	NO	NO	YES	YES	YES
BUILT-IN DATA RECORDING	NO	PARTIAL	YES	YES	YES
CONTROL FLOW					
o RULES FOR EXECUTION TIME DETERMINATION	NO	NO	YES	YES	YES
o DON'T CARE SEQUENCES	NO	NO	NO	YES	YES
FUNCTIONAL PERFORMANCE RQMT	NO	NO	NO	NO	YES
QUANTITATIVE PERFORMANCE RQMT	NO	NO	NO	NO	YES
PROCESS BINDING	NO	NO	SINGLE-PURPOSE	MULTI-PURPOSE	NO
LEVEL OF DIAGNOSTICS					
o SYNTAX	YES	YES	YES	YES	YES
o DP CONSTRAINTS	NO	NO	YES	YES	NO
o LOGIC	NO	NO	NO	YES	YES
o RQMTS INCONSISTENCIES	NO	NO	NO	NO	YES
o TIMING	NO	NO	NO	YES	NO
PRINCIPAL USER	PROGRAMMER	PROG.ENGR.	PROCESS DSGNR	PROCESS DSGNR	REQMTS.ENGR.
EXAMPLES	FORTRAN	SIMSCRIPT	PCL	PDL	NONE
	PL-1	GPSS			

Table 3.2 Candidate Language Requirements

DATA REDUCTION AND REPORT GENERATION

- Data Collection
  - Processing Macros
  - Recording Commands
- I/O Formatting
- Post Processing
- Report Generation
- Debug

SIMULATOR EXECUTION

- Data Set Initialization
- Test Case Selection
- Execution Job Step Control
- Exogenous Event Selection

SIMULATOR CONSOLIDATION

- H/W Config. Spec.
- Sys. S/W Config. Spec.
- S/W Module Selection
- OVLY Structuring Control
- Execution Control (Exec. init.)
- Time Modeling

FUNCTION MODELING

- User Defined Macros
- Executable Code (Arith & logical)
- "Data Set" Access
- "List Processing"
- Table Modeling
- Event Processing
- Time Posting

READABILITY/USABILITY

- Structured Prog. Conventions (Identation, if-then-else, do while, etc.)
- Short Form/Long Form Expansion
- Interactive or Batch Everywhere
- Identification, Reference, Retrieval by User Assigned Names

DATA BASE CONTROL

- Global DB Dictionary
  - Elements
  - Grouping
  - Indexing
  - Ownership
  - Coord. Sys/Units/Type/etc.
- "Function" I/O
- Functional/Analytic Data Conversion
- File Structuring Control

LIBRARY MANAGEMENT

- Creation, Update, Purge, ...
- Configuration Control
- Multiple, Hierarchical Libraries
- Security Controls

RQMTS SPECIFICATION

- System Requirements
  - Structured operating rules
  - System performance rqmts/ design goals
  - Assumptions List
- Misc
  - Design Constraints
  - Preamble data
- Interfaces
  - DP/External I/F's
  - Functional
  - Detailed

TESTING

- Test Type
  - "System" Test
- Mode
  - Static Validation
  - Functional
  - Analytic
- Test Case Generalization/Selection
- Test Invocation
- Test Report Level (Type)
- Driver Selection

Table 3.2 Candidate Language Requirements (Continued)

DOCUMENTATION GENERATION

- Media
  - Flow Charts
  - Graphics
  - Structured Text
  - I/O Devices
- Type
  - Specification
  - Test Results
  - I/F Diagrams
- Configuration Mgmt Controls

MANAGEMENT VISIBILITY REPORTS

- Critical Path Analysis
- Cost/Schedule
- Resources
- Responsible Party
- Traceability/Design Breakage
- Status
- Error Reporting/Analysis

decisions properly made by the process designer, and support a high degree of automation of the validation and documentation efforts.

TRW will develop and maintain close coordination with TI to insure this information interface is properly defined. As with the SRPS/Software Requirements Engineering interface particular attention will be directed at defining requirements on the CISS to support rapid, low-cost changes resulting from system changes.

Key technical issues to be resolved include:

- Methods of stating timing and accuracy requirements
- How to maintain computer-independence without requiring an infinite computer
- How to specify requirements on error processing/response when hardware is not selected
- Detailed examination of thread concepts evolved under previous SREP efforts.

The CISS analysis will include a study of the level of detail for the information set required by the process designer. This analysis will include the following:

- The intent of the CISS
- The lowest level of computer-independence
- The determination of completeness

The analysis will include a definition of the necessary and sufficient information content required, which includes the

- Functional Requirements
- Operational Requirements
- Performance Requirements
- General Design and Constraints
- Test Requirements
- Supporting Information

The analysis will determine a structure to present the information content. Structure elements may include:

- Language
- Block diagrams (Algorithm Sequencing Logic)
- Tables
- Figures
- Mathematics

The analysis will determine if the above is a complete set of information, and if not, shall determine what other information is required, such as:

- System Overview Documents
- Engineering Design Documents
- System Level Test Objectives

The analysis will determine a format for the information to reduce confusion between requirements and supporting information.

During the analysis a glossary of terminology will be developed to provide readability and reduce any misunderstanding.

The analysis will include an example, written in the CISS format, which contains the following properties:

- Two or more subsystems
- Analytical transformations
- Logical transformations
- Functional interaction

The analysis will attempt to define a measure of success to determine the readability and completeness of the computer-independent information set required by the process designer. This will be supported by one or more experiments, using the example in CISS format, to determine independently the quality of the information content and the form in which it is presented.

3.2.2.4 Definition of the Systems Engineering/Software Requirements Engineering Information Interface (SRPS)

A major objective will be to clearly define the interaction and information flow between the system definition activities and the software requirements analysis methodology. This will include a definition of the recommended contents and form of the System Requirements and Performance Specification (SRPS), which provides the basic initiating information with which the methodology begins. The requirements on this interface to support rapid, low-cost revisions to system changes or changes in system requirements impacting the data processing area will be defined.

Again, the approach to defining the SRPS contents will involve use of examples and small test cases to give confidence that the defined contents are realistically achievable. Technical interchange with GRC is planned to insure that a well-integrated interface will exist.

3.2.2.5 Baseline Methodology Synthesis

Based on the identified capabilities, analyses and design studies will be carried out to synthesize the recommended baseline methodology for the derivation and specification of software requirements. The desired capabilities and characteristics of the appropriate computer-aided requirements analysis and simulator construction tools to support the methodology will be defined.

Extensive research has already been conducted into the methods for specifying requirements of software that most aptly suit the BMD process. Real-time command and control dictate that the following must be provided in any set of requirements:

- The performance required of the software in terms of the accuracy of any desired response.
- The time that is available to produce the desired response.
- The processing (transformation) that is necessary to achieve the desired response.
- The stimulus (input) to the system and the conditions that exist in the environment that will result in the desired response given the necessary transformation.
- The arrival rates of the stimulus into the data processor subsystem.

These necessary conditions were used as a basis for developing a conceptual framework for a candidate methodology for deriving requirements. This candidate methodology is undergoing comparison with other methodologies that are currently available or under development. This methodology centered around stating the requirements in terms of stimulus, transformations, and desired response with each such set being called a thread.

Consistent with the incremental, empirically verified approach, the candidate methodology will be subjected to a series of examples to identify strong points, weak points, and areas that require additional research.

Identified strong points in the candidate methodology include the following:

- The requirements are stated essentially free of any design constraints. Unnecessary structures are not imposed on the process designer and processing that is independent of any order is identified.
- Discipline is introduced into the process of deriving requirements.
- The candidate methodology is amenable to automation and to the development of a requirements specification language.
- The use of the methodology produces requirements which are testable as a unit or as a system when integrated into a candidate system construct.
- Individual requirements can be traced at all levels of development.

Potential weaknesses in the candidate methodology were also identified so that additional research could be conducted to ascertain the true extent of these potential trouble areas. Among these were the following two:

- The number of stimuli, transformations, desired response sets may become so large on a complex real-time problem that handling them is externally difficult.
- The methodology for identifying the need for and requirements of a resource allocator has not been clearly defined.

3.2.2.6 Definition of Automation Requirements

This activity will identify that part of the methodology for which it is beneficial and feasible to automate or support with computer-aided tools.

The various computer programs involved are envisioned as a set of integrated tools which will support the engineering methodology, automate routine tasks, and enforce the central standards and collection of management information. Research to date has indicated that goals for the tools and the software requirements language include the following:

- Provide a unified support software system which provides cost-effective support to all phases of software requirements development.
- Provide a unified language for use in development requirements.
- Support configuration control.
- Provide management visibility information as an integral by-product.
- Provide maximum utility to the user by aiding him in the solution of his problem with minimum encumberances in the enforcement of configuration control and the collection of management visibility data.
- Automate as much as possible--especially simulations, documentation, test requirements, and test results.
- Provide means to establish positive correlation between specifications and simulations used to validate them.

The requirements for the tools will be developed by examining the identified tasks which comprise the methodology for the following:

- The way in which the task, or portions of it, could be automated or supported by computer-aided tools.
- The benefits of such automation.
- The relative difficulty of such automation.
- The preliminary estimate of cost to automate.

Several levels of automation of each task could be identified in this way. The results of this analysis will be evaluated to arrive at a full list of capabilities of a fully automated system, and possibly some intermediate automation points could be synthesized between it and a low cost/high payoff list of automated tools.

It is noted that, in order to achieve a low cost/high payoff system, the availability of current tools which implement or could be made to implement certain functions will be taken into account. For example, there will be a high payoff for the identification of functions which could be accomplished with the Process Design System, either current or proposed capabilities.

3.2.3 Computer-Aided Tools Development

In parallel and in conjunction with the definition of the software requirements engineering methodology and language, a system of automated aids will be designed to support use of the methodology. The objective for automating as many of the procedures in the methodology as possible is to reduce the ellapsed time and costs in producing a validated software requirements specification. Under this task, studies will be performed investigating feasibility of proposed automatable procedures, some involving implementation of prototype code. Once the requirements for and the feasibility of the procedures have been established, a preliminary design of the entire system will be developed. The actual implementation of the system will be accomplished in phases with a subset of the procedures identified as "critical to program success" developed in the initial phase. This subset of software components will be determined after the preliminary design and based primarily on cost/benefit analysis. Experiments conducted with the initial software components will demonstrate the components utility in the requirements specification process and provide requirements for both software and methodology revision.

3.2.3.1 Preliminary System Design

The requirements for automation will be determined in conjunction with the definition of the methodology capabilities and influenced by the techniques employed in the "proof of concept" experiments to be conducted. Additionally, requirements will be defined for general capabilities (methodology-independent), which support not only the methodology but the automated procedures development as well.

Automation requirements derived from the methodology are expected from, but not limited to, the following areas:

- Requirement Specification Language (RSL) Syntax Conversion to a Structured Requirements Simulation Language
- Data Base Specification and Structuring
- Algorithm Sequencing Logic Synthesis

General support requirements would be developed in the following areas:

- Configuration Management
- Library Management
- Simulation Data Base Management
- Report Generation
  - Simulation Results
  - Software Requirements Documents (i.e., CISS)
- Graphics

The definition of the RSL and the preliminary automated aids design will be closely coordinated to produce a language which is natural, requirements-oriented, and facilitates automation. In developing the preliminary design, use will be made of existing software capabilities to every extent possible. Alternate design approaches will be considered for all proposed software to investigate applicability of existing software. For example, in RSL processing, the following alternate approaches might be considered:

- Translate from an RSL to a base language offering the desired capabilities for simulation.
- Expand the capabilities of the base language to include the requirements language.
- Collect utility programs offering the desired capabilities and provide an interface to the RSL.

In each case the basic simulation capabilities might have to be extended. Selection of the design approach would be based on results of investigations into the applicability of existing software and support systems and possibly demonstrations of prototype code.

The preliminary design of the system will provide a description of the system capabilities, interfaces and a functional description of the support software at the major module level. The results of the investigations into requirements feasibility and prototype designs will also be included in the preliminary design.

3.2.3.2 Initial Phase Implementation

The preliminary objective of the initial implementation phase is to implement a subset of the total system which demonstrates methodology concepts and total system flexibility. The software will be designed specifically to support the requirements of, and collect applicable data for, the evaluation experiment. It will be designed to be usable in other applications and would be extendable to include additional system features. The criteria for selecting the initial capabilities to be implemented would include:

- Cost
- Schedule Limitations
- Visibility into Methodology Concepts
- Hardware Constraints/Limitations
- Availability of software with required capabilities
- Uniqueness of the procedure to be implemented
- Overall utility

Specific recommendations will be made to BMDATC for approval.

3.2.4 Methodology Experimentation and Evaluation

The motivations for experimentation in the SREP are methodology synthesis and evaluation. In supporting methodology synthesis, the primary experimentation objectives are to provide continual crystallization of and feedback into methodology concepts, and insight into the automatable procedures of the methodology and requirements on the language. To support methodology evaluation, experimentation will provide a data base for assessment of the methodology and for projection of methodology concepts to other systems development, a mechanism for continual evaluation of the overall methodology, and visibility into the methodology. These varying objectives require both informal and formal experimentation.

The informal experimentation, consisting of examples and test cases, will be conducted primarily during initial methodology synthesis. Sample problems will be used to apply methodology concepts, further refine concepts and procedures, and experiment with automation ideas related to synthesis and generation of simulations. Since these experiments are directed toward individual methodology concept extraction and practical application and preliminary automation feasibility assessment, methodology evaluation measurements will not be attempted.

In order to meet the objectives of methodology evaluation, formal experimentation is required. This will consist of a series of interrelated experiments leading to a large evaluation experiment to be initiated in FY 76. A preliminary experimentation plan will be developed describing the types of experiments to be conducted, measurement quantities and mechanisms, and experiment controls. This initial plan will be updated following methodology synthesis, tools capabilities definition, and determination of the initial tools capabilities. Initially, specific experiments will not be specified; selection of actual experiments will be subject to BMDATC approval.

In designing experiments, emphasis will be placed on collecting data related to the problem areas the methodology attempts to solve or alleviate. Measurement techniques will be devised to provide assessment data in such development aspects as cost, time, predictability and reliability of the resultant software, testability of specifications and software and learnability of the methodology. This requires hard measures such as cost, time, and history of errors. However, in order to elucidate these aspects additional information such as level of skills, knowledge of problem, and management participation must be obtained for different facets of the methodology.

Additionally, since the Software Requirements Engineering Methodology is an element in a total effort to develop a method to reduce overall software development and life-cycle costs, in order to effectively synthesize and evaluate the methodology and to provide the proper end product (CISS) to the process designer, continual participation of the process designer is essential. Efforts will be made throughout experimentation to obtain evaluation feedback and input from the process designer.

4.0 STUDY LIMITATIONS AND OTHER CONSIDERATIONS

The previously described research activities define a general overall plan to address the problem statement presented in Section 2. The top-level schedule shown is based on the present understanding of overall BMDATC program objectives and does not specifically consider cost and schedule constraints that will be defined in actual contracts.

The planned research will require technical interchange with and, in the case of the SRPS for Update 1, timely receipt of inputs from other BMDATC contractors.

The methodology and tools will be developed under the assumption that they will be utilized in the development of BMD-oriented real-time software. Even though the focus will be on BMD, the concepts and tools could be applied in the development of any large-scale software engineering project, and therefore, wherever feasible, generality will be maintained.

Because the goal of this research is to define a methodology and supporting computer-aided tools, full implementation, testing and user documentation of all the identified tools is not appropriate or envisioned within this plan. The results of these activities, however, will indicate the direction of a future program aimed at complete implementation of the integrated set of tools.

5.0 GLOSSARY

ASL	Algorithm Sequencing Logic
BMD	Ballistic Missile Defense
BMDATC	Ballistic Missile Defense Advanced Technology Center
CDPDS	Computer-Dependent Process Design Specification
CIS	Computer-Independent Specification
CISS	Computer-Independent Software Specification
CIS-SFR	Computer-Independent Specification of Software Functional Requirements
GRC	General Research Corporation
PCL	Process Construction Language
PCP	Process Construction Program
PDL	Process Design Language
PDS	Process Design System
RSL	Requirement Specification Language
SREP	Software Requirements Engineering Program
SRPS	System Requirements and Performance Specification
STATIC TESTING	That portion of software testing which can be accomplished prior to execution of that software, e.g., I/O consistency between elements of thread.
TDP	Terminal Defense Program
THREAD	The sequence of processing steps within a subsystem which is initiated by a stimulus at an input port and results in the appropriate response at an output port or internal process termination point.
TI	Texas Instruments

6.0 REFERENCES

1. Boehm, B. W., "Software and Its Impact: A Quantitative Assessment," Datamation, May 1973.

2. McGonagle, J. D., "A Study of a Software Development Project," James P. Anderson & Co., Sept. 1971.

3. Bartlett, J. C. et. al., "Software Validation Study," Logicon, Inc., March 1973.

4. Herring, F. P. et. al., "Current Software Requirements Engineering Technology," TRW Systems Group, August 15, 1974.

22944-6921-015

SOFTWARE CAPABILITY DESCRIPTION - SOFTWARE REQUIREMENTS ENGINEERING METHODOLOGY

CDRL GOOB

OCTOBER 1, 1974

Sponsored By
BALLISTIC MISSILE DEFENSE
ADVANCED TECHNOLOGY CENTER

DAHC60-71-C-0049

TRW
SYSTEMS GROUP
Huntsville, Alabama

SOFTWARE CAPABILITY DESCRIPTION-SOFTWARE REQUIREMENTS ENGINEERING METHODOLOGY

CDRL G00B

OCTOBER 1, 1974

Sponsored By
Ballistic Missile Defense
Advanced Technology Center
DAHC60-71-C-0049

TRW
SYSTEMS GROUP
Huntsville, Alabama

SOFTWARE CAPABILITY DESCRIPTION-SOFTWARE REQUIREMENTS ENGINEERING METHODOLOGY

CDRL GOOB

OCTOBER 1, 1974

Principal Authors

M. Alford
F. Burns
M. Hansing
J. Richardson
R. Smith

Approved By:

T. W. Kampe

T. W. Kampe, Manager
Software Requirements
Engineering Program

James E. Long

James E. Long, Manager
Huntsville Army Support Facility

Sponsored By
Ballistic Missile Defense
Advanced Technology Center
DAHC60-71-C-0049

TRW
SYSTEMS GROUP
Huntsville, Alabama

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

1.0 INTRODUCTION

The Ballistic Missile Defense Advanced Technology Center (BMDATC) has established the Software Development and Evaluation Program in order to develop a complete and unified engineering approach to software development and testing ranging from synthesis of system specifications through completion and testing of the process design. A key element of this program is the Software Requirements Engineering Program (SREP)--a research program concerned with the development of a systematic approach to the development of complete and validated software requirements specifications. Primary objectives [1] of this research are to ensure a well-defined technique for the decomposition of system requirements into structured software requirements, provide requirements development visibility, maintain requirements development independent of machine implementation, allow for easy response to system requirements changes, and provide for testable and easily validated software requirements.

To meet these objectives, the Software Requirements Engineering Methodology (SREM) is being developed. This methodology is an integrated, structured approach to the requirements engineering activities from the parsing and translation into data processing software requirements of system requirements provided in a System Requirements and Performance Specification (SRPS) through analysis, refinement, validation and documentation of the software requirements in a Process Performance Requirements (PPR) Specification.

This report documents the capabilities required in the SREM and forms the basis for a more detailed design of the SREM including the refinement of the engineering techniques already identified; the definition of a structured medium, or language, for specifying requirements; the definition and development of integrated software tools to support requirements development; and the identification of the detailed procedures for their application. Section 2 states the problems which the methodology must address. Section 3 discusses key assumptions concerning the types of systems to which the methodology is to be applied. The fundamental concepts which form the basis for design of the methodology are presented in Section 4.

Section 5 identifies the capabilities which the engineering techniques, language, support tools, and detailed application procedures must provide.

The contents of this document conform to those specified in the ABMDA Research and Development Software Standards [3], Software Capability Description, where applicable to the capabilities of a methodology. This document, therefore, is not intended to present the results of all research to date. Previous reports in this series of documentation include: an evaluation of current techniques [4]; a research plan [1]; language requirements [2]; and an experiment plan to validate the methodology [5]. Present plans for the Software Requirements Engineering Project call for reports to be published describing: The form and contents for a SRPS and a PPR in January, 1975; a detailed description of the methodology, with examples, in March, 1975; a preliminary design of all software tools to be produced in May, 1975; and the tools to be available for demonstration in December, 1975.

2.0 PROBLEM DESCRIPTION

Examination of software development problems and their symptoms has shown a strong connection with, and a definite need for improvement in, software requirements engineering. The initial problem statement for the overall methodology research [1] discusses these problems and their relationship to requirements engineering and serves as a starting point from which the desired methodology capabilities can be identified. That problem statement has been further analyzed and expanded to identify those problems which the methodology must address. This expansion has been divided into two categories: 1) those relating to the process of obtaining a software specification, and 2) those involving the properties of the resulting specification itself. Both are summarized below.

2.1 PROBLEMS OF THE SPECIFICATION PROCESS

Although the problems of specifying software requirements discussed in [1] are interrelated, they basically involve five key issues.

1) Management of the requirements development process.
In general there has not been a well-defined approach with visible intermediate results in the development of software requirements. The measurement of requirements generation progress and the relationship between system engineering, software requirements engineering, and software design has been unclear. Lack of well-defined intermediate design levels has made software feasibility assessment and intermediate requirements validation difficult. Because the process was not well-defined, management of these activities has been difficult.

2) Communication interaction with System Engineering and Process Design.
Generally, methods of requirements communications between development levels have been ambiguous and imprecise, have implied or imposed design selections inappropriately, and have allowed incomplete and internally inconsistent specifications to result. Requirements statements which are untestable have been common. Design freedom to meet constraints has not been clearly identified.

3) Validation of requirements.
Requirements may be precise, complete, consistent and still be invalid; i.e., if met, will in fact, not make the system function as desired. Invalid requirements are possibly more serious than incomplete or inconsistent requirements, in that with the latter software design will generally expose the problem, but with

invalid requirements erroneous software will be generated. Generally, requirements validation is largely ignored, with most efforts on validation being focused on the final software. At best, there is an attempt to achieve validation with one single design solution by simulation. Although this may show there is "a" solution to the requirements, it does not prove that any design which meets the requirements will satisfy the system problem.

4) Time and degree of manual efforts required for requirements development. For large complex systems, a real problem is that the time required to implement known techniques, such as simulation for validation, is prohibitive. The difficulty of keeping documentation and simulators together and the potentially large number of human errors made during analysis all can combine to reduce the accuracy and reliability of the requirements effort. These problems are particularly troublesome in responding to changes in a timely manner.

5) Feasibility of the system.
It has been recognized that a requirements specification should state "what" is to be done, when, and how well; but not "how" it is to be accomplsihed. A good requirements statement should allow a maximum of freedom in the subsequent design and implementation phases. If it over-constrains the design, it will at least result in obstructing the designers ability to find the most efficient or most effective implementation of the real requirements. At most, it can generate inconsistencies and lead to resolutions in which the valid underlying requirement is compromised.

The concept of computer-independence results, in part, from this desire to avoid implementation assumptions when developing requirements. However, it must be recognized that the requirements development process is part of an overall system design process and that design decisions are, and must be, made. Therefore, feasibility must be considered at all levels, else, the process may proceed down costly impractical paths. The requirements development process should, therefore, provide data, support a growing confidence that the system is feasible, and consider potential feasibility problems when making design decisions.

2.2 SPECIFICATION PROBLEMS

Analysis of the problems and issues [1] has led to an understanding of the necessary attributes of a good requirements specification. The requirements development process must be capable of producing specifications which address the three major areas which are summarized below.

1) Correctness of requirements.
This covers all aspects of the statement of the performance which must be attained. This includes:

- Completeness. "What you see is what you get" is the rule. If a capability, feature, or performance parameter is not specified as a requirement, there is absolutely no reason to believe that it will appear in the final product. Engineers and programmers are typically honest and professional, but they are subject to schedule and budget constraints. Implementing things that are not specified is poor management on their part. Therefore, it is imperative that the requirements specification must contain everything expected of the system.

- Testability. It is obvious that the system should be tested for conformance to the specification to which it was built. However, it is surprisingly simple to write requirements which look very good only to discover later that there is absolutely no way to test the end product for compliance. Every requirement specified must be examined for testability. If it is found to be untestable or unverifiable, it should be changed.

- Explicitness. This attribute is a corollary to completeness and testability. The requirements must be stated explicitly. The specification should not require the reader to "read between the lines," correlate two statements to obtain an implied requirement, or to otherwise apply analysis to discover what is required. Additionally, all terms used must be unambiguous. The law of perversity guarantees that if two meanings can be applied to a statement in a specification, the wrong one will be followed.

2) Design freedom.
The software designer must be told what degrees of freedom are available to meet the constraints. This includes:

- Design Independence. A requirements specification states "what" is to be done, when, and how well, but not "how" it is to be accomplished for real-time software. A good requirements specification should allow a maximum of freedom in the subsequent design and implementation phases. This does not imply that design decisions are not made in the development of the requirements--they are. But, no design decision should be arbitrarily made which unnecessarily restricts the design freedom of the next man in the development cycle. This means that the techniques, formats, and means of presenting the requirements must not inadvertently introduce unintentional design choices.

- Sufficiency. A requirements specification must not only state everything which is required of the system, but must also supply the information needed by the designer to do his job. Information known to the requirements engineer should not be left for the designer to re-invent or rediscover. Information which would be useful to the designer and does not logically fit into the specification itself can be included in a for-information-only appendix or in separate documents.

3) Bookkeeping.
This covers all attributes of a specification which address the fact that it is a part of a family of specifications. This includes:

- Traceability. In a large system, several levels of requirements and design specifications exist. Upwards and downwards traceability must exist. Downward traceability allows one to verify that every requirement in a specification has been considered in lower level documents and allows identification of where a change in requirements affects design. Upward traceability answers the question "why is that being done?" It allows verification of performance against the parent requirements and allows an impact analysis to be made in the event that a detailed performance requirement cannot be met.

- Modularity. Requirements should be modular for the same reasons that the software should: (1) change is to be expected as a way of life. If the requirements are modular, then changes are easier to analyze, invoke, and control. (2) As the details get filled in and the total volume of material and work grows, division of labor becomes a must. Modularity allows a rational division of labor. Useful modularity means that each "module" of requirements be internally complete and that it fits into the entire system through very well defined (and controlled) interfaces with other "modules."

3.0 KEY ASSUMPTIONS

The Software Requirements Engineering Methodology under development to handle the problems of the last section is influenced by assumptions concerning its scope and the nature of the systems to which it is to be applied. These are discussed in the following sections.

3.1 SCOPE OF ACTIVITIES IN THE SYSTEM DEVELOPMENT CYCLE

The range of activities which are to be addressed by the Software Requirements Engineering Methodology requires both a knowledge of systems engineering and of data processing technology. On the systems engineering side, the range of SREM starts when data processing expertise is required to carry out systems engineering analysis. On the data processing side, the SREM activities cease when the knowledge of system engineering is no longer required.

To elaborate, the starting point of SREM (see Figure 3-1) is the point in systems engineering when the system requirements analysis is well underway, the function and the stress points of the system are known. The interfaces between the subsystems have been established (at least on the functional level). The top level system functions have been identified and a set of system operating rules (conditional statements impacting when and in what sequence the functions are performed) have been established, although probably incomplete. The top level system functions have been allocated to the subsystems, including the data processor.

The ending point of SREM is that point when all system requirements allocated to the data processor have been sufficiently defined in data processor terms so that computer science is the primary expertise required to continue. The DP interfaces have been determined to the element level, all processing steps have been identified with appropriate DP requirements levied, and all actions of the DP in response to a stimulus are determined in terms of sequences of processing steps.

3.2 COMPUTER INDEPENDENCE

A restriction on the resulting software specification is that it be computer-independent, i.e., not specifically restricted to a specific data processor or data processing architecture type. This does not mean that an "infinite" computer should be assumed; "computer-independent" measures of data processing activity (e.g., processing capability, interface data rate, data storage required, time responses required, etc.) are to be assessed in order to determine preliminary feasibility. If system or DP subsystem performance is sensitive to the performance of a specific algorithm, this is to be identified and the results are to be brought to the attention of Systems Engineering. As previously stated in Section 2.1, the concept of computer-independence results, in part, from this desire to avoid implementation assumptions when developing requirements. However, it must be recognized that the requirements development process is part of an overall system design process and that design decisions are, and must be, made. Therefore, feasibility must be considered at all levels, else, the process may procede down costly impractical paths. The requirements development process should, therefore, provide data, support a growing confidence that the system is feasible, and consider potential feasibility problems when making design decisions.

3.3 NATURE OF SYSTEMS ADDRESSED

The methodology is based on assumptions about the nature of the systems to which the methodology will be most advantageously applied and the system development environments in which it will be applied. The major system characteristics assumed are summarized by the following statements:

- Systems are large to very large;
- Time responses are critical;
- Processing is intensive;
- Data base is large but not massive;
- Computer system is digital with memory;

- Technology of the object system is not fully understood initially (i.e., existence and feasibility of the system and subsystems is an issue);
- Requirements are subject to a high degree of change;
- Subsystems interfacing with the DP may (probably) be undergoing parallel development (i.e., interfaces are flexible).

System development environments are assumed to be of two types, namely:

- A deployable system being developed essentially top-down from system performance objectives, which must deal with
  - Hard performance requirements,
  - Firm threat definition,
  - Strong top-level configuration control,
  - Slow change control,
  - Maximum design freedom, and
  - Cost and schedule as major considerations.
- R&D project where design decisions are influenced by objectives of investigations specific approaches (e.g., improvement of the state-of-the-art) which must deal with
  - Minimum performance requirements,
  - Flexible threat,
  - Less-formal configuration control,
  - Reduced design freedom, and
  - Cost and schedule of lesser importance.

3.4 THE ROLE OF SETS

Because simulator construction and utilization play such a key role in the Software Requirements Engineering Methodology, the role of a System Environment and Threat Simulator (SETS) must be clarified. For the purposes of the following discussion, it is assumed that there are SETS development activities independent of SREM. However, it is assumed that SREM must support the establishment of the interface requirements between SETS and the data processing subsystem, to determine the required information for the validation

of the requirements and the validation of the resulting real-time data processing subsystem. This is required because the validation of requirements may require additional interface information to perform tradeoff and sensitivity studies not required by the real-time software. It is noted, that because SETS is itself a major real-time software package, SREM could be advantageously applied to its development as well as to most other test-support software.

3.5 NON-PROCESSING REQUIREMENTS

There are some DP requirements issued at the system level (e.g., maintainability, documentation standards, hardware-related aspects, security, etc.) which are non-functional and inherently untestable to a precise degree and as a result, the methodology offers little analysis. However, its bookkeeping capability should provide a place for the statement of such requirements in order to maintain continuity of specifications.

4.0 FUNDAMENTAL CONCEPTS OF THE APPROACH

A Software Requirements Engineering Methodology which addresses the previously discussed problems and assumptions should provide the following attributes:

- A structured approach to developing software requirements which,
  - defines techniques for decomposition of system requirements into software requirements which have the identified attributes of a good requirements specification,
  - improves management visibility through intermediate products and milestones,
  - addresses requirements validity at all levels, and
  - supports a continuing assessment of system feasibility without imposing improper design decisions on the software designer.
- A structured medium, or language, for specifying and communicating requirements to the process designer which
  - reduces ambiguity,
  - supports traceability, and
  - supports automation.
- Integrated use of computer-aided tools to assist in the requirements development efforts which
  - maintain an information base,
  - reduce the development time for requirements, and
  - reduce human error.

To design a methodology with such attributes, a set of fundamental concepts are exploited to achieve an approach for structuring and stating requirements, to provide a means for feasibility analysis, and to determine the existence of algorithms. These ideas are discussed below.

4.1 STRUCTURING REQUIREMENTS

A fundamental idea which the Software Requirements Engineering Methodology exploits is that software performance requirements can easily be stated in terms of processing steps and sequences of processing steps in a manner which is testable. This allows maximal software design freedom, which addresses traceability to both the system and software design, and provides management visibility.

The form and content of a software specification is the implicit result of a tradeoff between testability, design freedom, and bookkeeping considerations. The testability considerations restrict the form and content of a software specification by dictating that each requirement be explicitly testable, i.e., without knowledge of the specific software design, one ought to be able to devise a test with explicit pass/fail criteria to determine whether the software meets that particular criteria. Software (and sometimes hardware) design freedom considerations dictate that the process designer be given appropriate degrees of freedom to meet the testable constraints; the more demanding the constraints, the more it is necessary that the constraints themselves be formulated in such a way that involves minimal impact on the software design. Finally, a software specification must: be able to be modified easily; be traceable to the system design, software design, and testing; and be in such a form that software versions can easily be identified. All of these are "bookkeeping" considerations which restrict the form of the specification.

The underlying approach is to state all possible software requirements in terms which are testable by observation only of data at the data processor interfaces. This includes system timing requirements and interface requirements such as radar timeline constraints. These requirements are to be associated with sequences of processing steps, and with the conditions under which the constraints must hold. Stated in this form, the requirements are explicitly testable, and allow maximal design freedom for the process designer (i.e., any software design which accomplishes these processing steps with the required accuracy is allowable, including choice of design structure and of algorithm approach). Moreover, the many diverse bookkeeping considerations are addressed: Capabilities of the software may be added and deleted without

modification of an existing set of requirements; versions of the software may be identified by reference to the set processing sequences, their conditions and requirements.

There exist requirements, however, which cannot be explicitly described or tested using only interface information. These are to be described using processing sequences, and their constraints are to be expressed in terms of information available at explicitly identified "test points" in the sequence of processing steps. These can be in terms of data which must be available in some form in the data processor data base (e.g., global data base), or even in terms of all inputs and outputs to a specific algorithm. These requirements are to be stated in a form which allows maximal design freedom by identifying the maximum number of processing steps between test points, but still allow the requirements to be explicitly testable in terms of those test points. This format is flexible enough to include the range of requirements from port to port, or to requirements specific to an identifiable algorithm. This allows a uniform bookkeeping mechanism to be used to keep track of all of the requirements from coding requirements for a specific algorithm, to port-to-port timing requirements, to interface requirements. Similar requirements on processing sequences are grouped together into "trees" of sequences in order to simplify the bookkeeping of those requirements and to simplify the checking for completeness and consistency.

Requirements organized in this manner are the basis for the automatic generation of simulators: Both functional simulators (which are used to address data processing feasibility issues such as instruction execution rate or data interface requirements) and analytical emulators, which include exemplary algorithms for all processing steps in order to validate the software performance requirements. This addresses the issues of both simulator traceability and timeliness.

Finally, the state of development of the requirements (e.g., stated only in systems performance terms, stated in data processing terms, exemplary algorithm exists, requirement has been validated) can be organized in these terms and hence provide management visibility into the degree of completion of the requirements analysis effort. This addresses the required capability of the methodology to enhance the process of obtaining a set of software requirements.

4.2 FEASIBILITY ANALYSIS

A second major focus of SREM is that of assuring the feasibility of the data processing subsystem. A top-down design methodology addresses a system design in terms of system parameters and then allocates performance to subsystems. Although this approach allows system feasibility to be addressed explicitly during the design process, there is always some doubt as to whether the requisite subsystems can be designed to their required performance levels. A necessary part of the SREM activities is to assure that the data processing subsystem requirements are feasible; this is the primary purpose of the simulations.

Simulators (or emulators) are built and used for two different purposes: To establish performance requirements; and to validate a given set of requirements. The use of simulation to validate requirements implies that there be a high degree of traceability between the requirements and the simulator or emulator. The use of a simulator as a design tool to establish feasibility of a set of requirements (e.g., demonstrate the performance of an algorithm), or to establish a set of accuracy and timing requirements such that system performance allocated to the data processor is met, implies that the simulators be constructed within a short time frame.

Finally, there are two types of simulators to be built: Functional simulators, by which system performance parameters are translated into data processing design parameters; and emulators, which include exemplary or prototype algorithms to set and validate performance requirements feasibility.

4.3 ALGORITHM SPECIFICATION

The crucial issue which algorithm analysis must address is that of feasibility: The existence of an algorithm to perform a specified computational task supports the feasibility of a data processing subsystem which can meet those requirements. Not only is the existence of a possible solution to be demonstrated, but its availability can be used in order to assess issues of computational sanity or completeness and of computational costs (e.g., MIPS, data base activity, storage costs). Note that, even though the specification activities are computer-independent, a computational value system can be

utilized in order to assess (at least in preliminary terms) whether any data processing subsystem which addresses all of the requirements is feasible. Candidate algorithms for critical processing steps provide confidence for such feasibility estimates.

Algorithms can be used in a specification in one of three different ways. In the least constraining case, an algorithm may be provided as an exemplary solution to the actual software requirements and is used purely for validation purposes (e.g., a radar scheduling algorithm which demonstrates that the desired pulse mix can in fact be scheduled to meet interface requirements). In other cases, a candidate algorithm may be created to be used as a standard of reference (e.g., a resource allocator should yield an allocation which matches the performance of the standard within 1%). And finally, in R&D types of environment, or in special cases where the determination of any performance bounds is difficult or involves explicit system tradeoffs, an algorithm may be provided with requirements that any solution be "as close as possible" to that provided (e.g., eight significant figures, or coded term by term). The Software Requirements Engineering Methodology is to be capable of expressing requirements on algorithms of any of the above types.

5.0 METHODOLOGY CAPABILITIES

This section presents a description of the steps necessary to produce a validated software specification from a SRPS. Although a discrete sequence of steps is described, most of the steps continue, once initiated, until the requirements definition activities are complete; in addition, for clarity of presentation, all feedback cycles have been omitted. For example, an algorithm development activity might discover that the SRPS Performance Requirements were infeasible; this might necessitate a change in the SRPS and subsequent documentation. Many of these steps are familiar, but this methodology formalizes, structures, and defines their outputs to make progress more visible.

The steps of the methodology are first discussed in an overview, a discussion of management visibility is given next, then each major step is described in more detail with a summary of the capabilities following.

5.1 METHODOLOGY OVERVIEW

An overview of the techniques and procedures of the methodology is presented in Figure 5-1. Basically, there are six major steps, each with measurable output. First, the SRPS contents are analyzed to identify holes, ambiguities, and conflicts with respect to software requirements. Based on this information, and appropriate feedback from systems engineering, the Data Processing (DP) Subsystem Capabilities are described, providing a baseline for subsequent requirements analysis. Two primary issues are important at this stage: 1) traceability is established from SRPS and 2) the sufficiency of the information content of SRPS is determined. This step provides a startup procedure for the methodology and top level DP requirements are obtained. The next two steps are directed at refining and validating these requirements.

The second step is the development of a software performance model and an Algorithm Sequencing Logic (ASL). A complete mapping is made between system level requirements (e.g., interceptor control loop requirements) and data processing level requirements (e.g., port-to-port time budget from receipt of interceptor beacon return until a guidance pulse command based

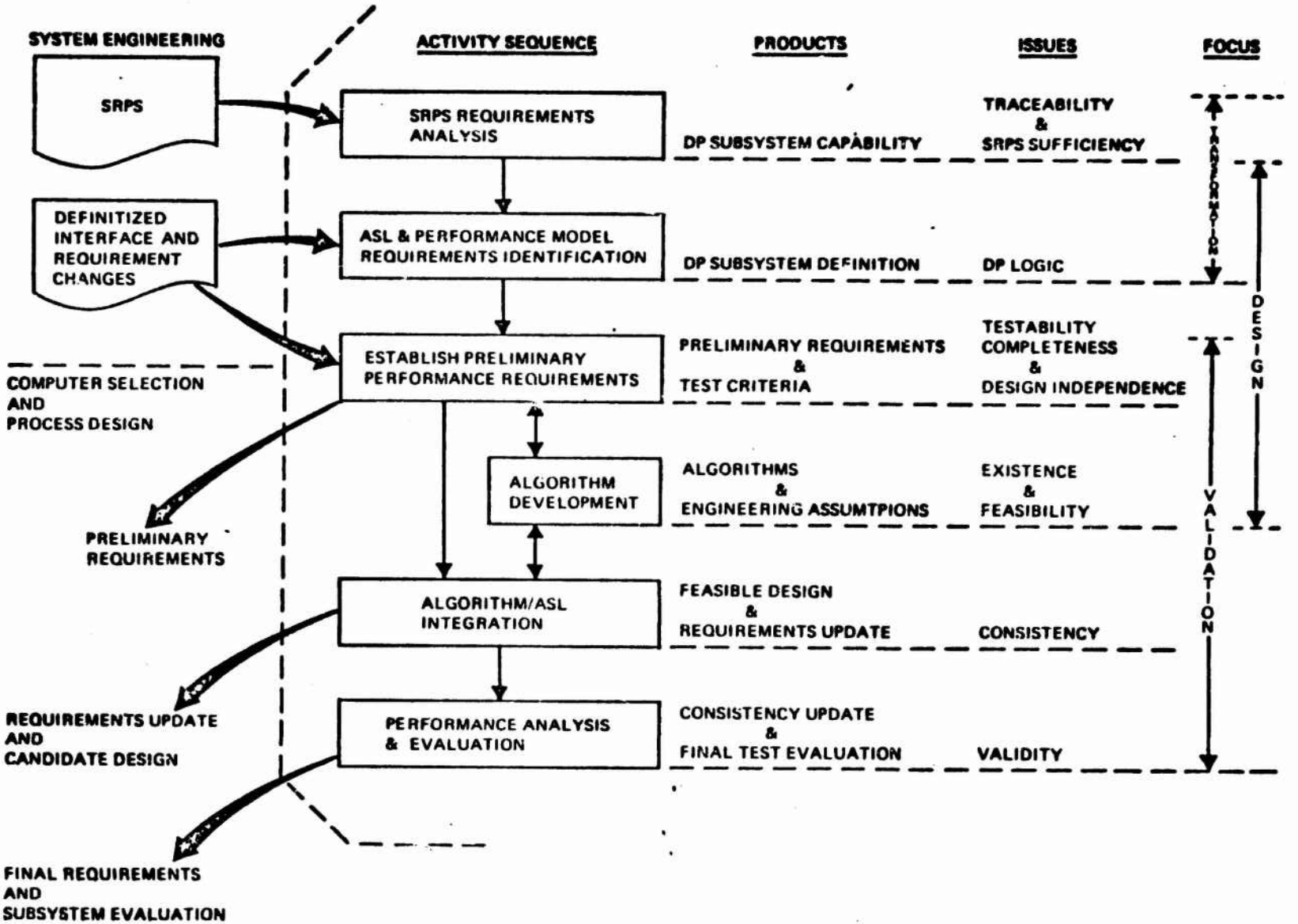


Figure 5-1 Top Level Methodology Description

on that data is sent to the radar). The emphasis at this step is on defining the data processing logic and establishing a transformation between DP allocated system requirements in the SRPS and the DP subsystem requirements in DP terms. The DP design process begins here and major interactions transpire with system engineering to obtain more definite interface specifications and requirement changes.

In the third step, a set of preliminary performance requirements and test criteria are developed. This phase requires a continuing interaction with system engineering regarding interface definition and requirements clarifications. The software performance model in step two is a major input into this activity. The result of this step is a Preliminary Requirements and Test Criteria document, which sets forth performance requirements on all algorithms and control software. The central focus of these activities is an assurance: That the requirements specified are testable: that they are stated in terms which allow appropriate software design freedom; that all of the requirements have been identified; and that the system performance has been correctly allocated to data processor performance. A functional simulator is necessary to establish the consistency of the definition of the processing steps, and the interfaces are checked against the processing definition to assure the specification is complete. The interfaces with SETS and other test control software are defined and integrated into the interface specifications. The requirements may not be feasible at this stage, or complete. This information is the basis for algorithm design and selection. This point in the methodology is viewed as the earliest possible time that the computer selection and process design phases can begin.

The fourth step addresses algorithm development. Starting with the description of each processing step in the software specification, the state-of-the-art in algorithm design is examined to determine existence and feasibility. In the event a particular specification admits to more than one feasible algorithm which meets all performance requirements, the algorithm designer should select one based on the estimated computational efficiency. This freedom is granted the algorithm designer so that any criteria beyond that in the specification may be considered. The central issue addressed is that of algorithm feasibility, e.g., does an interceptor planning algorithm exist which satisfies the system objectives?

In the fifth step, performance is re-allocated to handle better-than-expected or worse-than-expected algorithmic performance, once these are known. The algorithms are integrated and consistency of system actions are examined. At the end of this step there is a candidate solution which satisfies the requirements. The most significant results of this step are: A working data processor emulator, which can be used to validate the software performance requirements and to perform tradeoff studies; and a validation of the consistency of the requirements.

The final step is to check for completeness by detailed simulation. At the start of this step, analysis has done everything possible to assure completeness. An estimate of performance is made before running the test cases. If that estimate is proven correct by the test cases, one has heightened confidence in the prediction of performance for cases not run and for alternative design solutions. Output from this step is the collection of validated requirements along with the subsystem evaluation.

Requirements validation is a major component of this methodology and is a focus in all steps below step two. The following sections will expand on the overview presented above.

5.2 MANAGEMENT VISIBILITY

Management visibility of any activity is accomplished by the use of two different mechanisms: The identification of discrete milestones, which can be scheduled and reviewed for satisfactory progress; and the identification of progress indicators which can be used to reliably indicate the degree to which satisfactory progress of an activity has been made. The existence of either of these mechanisms has been lacking in previous software requirements methodologies.

It can be argued that, given any version of a SRPS, there is a milestone when the SRPS Requirements analysis has been completed and results reviewed with the Systems Engineering organization. However, it is well known that all requirements need not be specified in order to develop algorithms and carry out preliminary feasibility demonstrations of some of the more critical

processing steps. For example, in Site Defense, Discrimination was observed to be a critical issue and algorithm development work was initiated long before the detailed requirements of lost track processing. It is the nature of the systems engineering task to attack first those requirements which are most critical and to postpone consideration of the less critical requirements. Thus, it can be argued that there are no clear major milestones from the point where the SRPS requirements analysis begins and the software requirements have been generated and validated. At any point in between, there may only be a percentage of the requirements which have been analyzed to any specific depth, while others are already at a lower level of detail.

Figure 5-2 presents a model of the software requirements development cycle in terms of the SREM overview. The vertical axis represents the depth of analysis of a specific requirement or set of requirements (e.g., bulk filtering, maintenance tracking, error detection of the radar interface, etc.), while the horizontal axis represents the capabilities, ranging from most critical to least critical, of the final software specification.

When the SRPS analysis is complete, perhaps less than 20% of the data processing detailed requirements are known. As the SRPS is modified, the SRPS Requirements Analysis activity must be performed on the specification updates as they occur. When sufficient analysis has been done to identify enough data processing requirements to have a meaningful document (say at time t_2), perhaps less than 50% of the preliminary requirements may be known; detailed performance requirements may have been established for some capabilities, and some algorithm development might have been started to establish feasibility of the most critical software capabilities.

By the time that a first version of a Software Specification can be established (say time t_3), some algorithms may have already been validated while other capabilities might not have been yet identified (e.g., communication subsystem interface protocol). By the time that most of the algorithms have been identified for development (say time t_4), the performance requirements may not have been specified or even identified for many others. One can conceive of a requirements validation simulator at time t_5 which validates

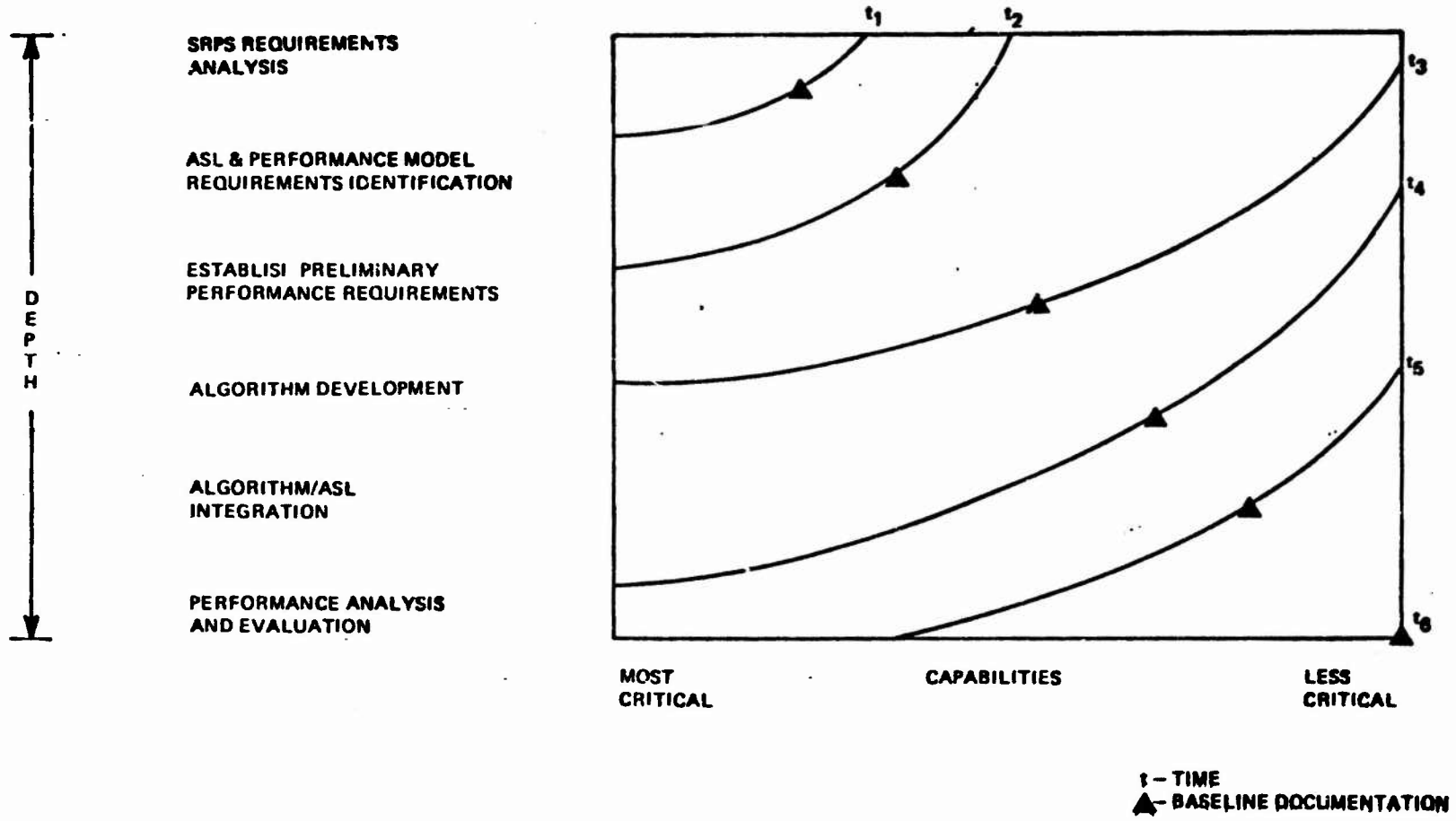


Figure 5-2 Development Cycle

a portion of the requirements while some algorithms have yet to be integrated. Finally, at some time t_6 one would hope to find that all requirements have been validated (in practice, this never happens because of augmentations to the SRPS until the final software is about to be delivered, if then).

In this process for the specification of the set of software requirements as a whole, there appear to be a set of progress indicators for each requirement or set of requirements separately. These indicators are the answers to the following questions:

- Have the requirements for the capability (e.g., thread or processing step) been identified?
- Have the requirements been identified in data processing terms?
- What is the estimated system criticality of the performance (in systems terms, e.g., leakage) to other subsystems?
- What is the estimated impact to the data processing subsystem performance (e.g., MIPS, data rate)?
- How much more systems engineering effort and schedule is necessary before the performance issues are resolved?
- How much more software requirements engineering effort and time is necessary before the data processor performance is established?
- Has a candidate algorithm been identified? If not, what is the effort and schedule to complete?
- Have the candidate algorithms been integrated? If not, what is the effort and schedule to complete?
- Have the candidate algorithms been included in the validation emulator? If not, how much effort and schedule is required to complete and run the test cases?

The degree of completion of each capability and the amount of effort to achieve the next milestone yields a great deal of information on both the progress achieved and efforts remaining for the requirements development as a whole.

This same approach can be used to evaluate the impact of changes to the SRPS on the production of the software requirements.

5.3 SRPS REQUIREMENTS ANALYSIS

The SRPS requirements analysis addresses the transformation from the system requirements to the software-oriented Data Processing requirements. The SRPS Specification states the technical and mission requirements for the system as an entity, allocates requirements to functional areas, and defines the initial interfaces between subsystems. These requirements, measured in terms of system effectiveness (leakage, interceptor miss distance, survivability, etc.), need to be transformed to some discrete quanta which can be measured and tested in the Data Processing Subsystem, independent of System Testing activities.

Generally, the types of information received in the SRPS are the functional things to do and how well they should be performed (maintain track such that the leakage does not exceed two percent). Operating rules which are the conditional statements impacting when and in what sequence the functions are performed (perform discrimination after the tracking accuracy is within five percent of nominal or the object has reached 100,000 feet); and, the various design constraints on the requirement to do something a certain way (do not exceed a radar peak power of 400 kilojoules). These requirement statements may require a great deal of analysis to determine what they mean in terms of Data Processing performance; more analysis is necessary to obtain an integrated set without ambiguities and conflicts. Figure 5-3 presents a more detailed sequence of activities which carries out such analysis.

The first step is to identify and classify requirements into functional categories and to enumerate them for traceability. This activity analyzes the information content of the SRPS, lists each requirement independently, and provides it with its own identifier to be used throughout the software requirement development cycle.

The second step, performed concurrently with the first, is to perform a Data Processing Input/Output Analysis. This activity determines the information content flowing to and from the Data Processor and identifies stimulus/response linkages between the input and output data.

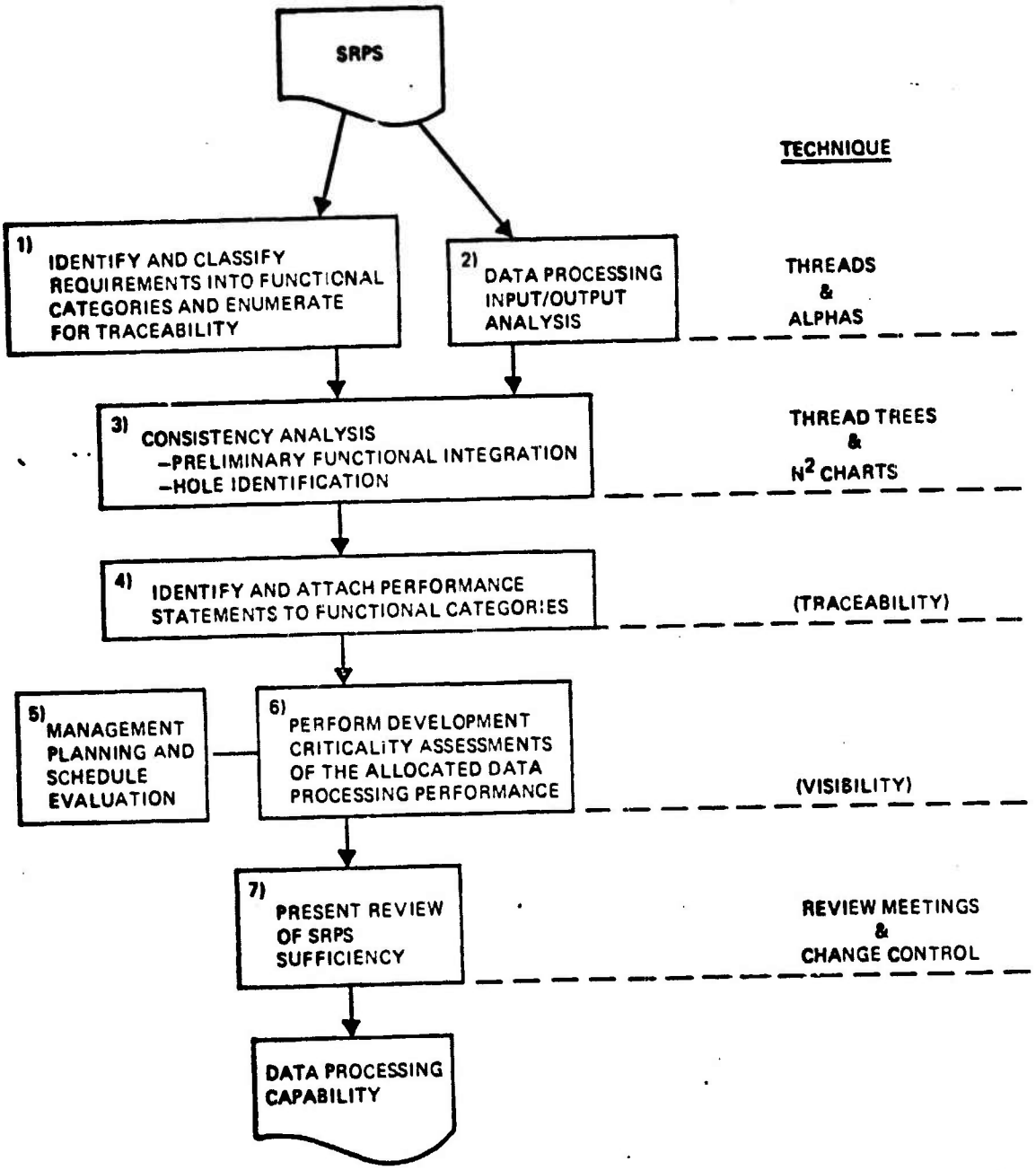


Figure 5-3 SRPS Requirements Analysis

The third step is a consistency analysis. This activity integrates the first two activities into one process, and identifies any ambiguities and conflicts with respect to software and uncovers functional holes. The next step is the identification and allocation of performance statements to functional categories to provide traceability to the SRPS. This step is complete only when all SRPS performance statements are accounted for.

The fifth step is the assessment of development criticality of the allocated Data Processing performance. This activity identifies Data Processing analysis studies, issues for system engineering resolutions, and those items which are critical to either system performance or subsystem feasibility. This complete analysis is presented to systems engineering for review to determine SRPS sufficiency for the future activities.

5.4 ASL AND PERFORMANCE MODEL REQUIREMENTS IDENTIFICATION

The second phase of the methodology, illustrated in Figure 5-4, identifies the operational and functional requirements which must be met by the DP Subsystem. The emphasis at this stage is on the data processor logic, a transformation/mapping of system level requirements into data processing level requirements, and identification of derived functional requirements. Primarily, this phase defines the functional data processor/software processing requirements which must be satisfied for the overall system to meet the SRPS requirements. The functional requirements are then assigned preliminary performance requirements in the third phase as bases to begin algorithm development. As an objective, this phase should identify all threads through the DP, with the exception of those which may be required as a direct response to computer-dependent conditions and requirements (such as computer hardware failure/error recovery requirements).

The first step during this phase is to refine and expand the functional requirements (stated in threads of processing steps) to a lower level. This expansion first reflects increased information resulting from a concurrent analysis (step 2) of the functional interfaces between the DP and other subsystems. The interface knowledge during the previous SRPS Requirements Analysis phase will be limited (e.g., only the type of radar return may be identified). Further analysis leads to the knowledge of major differences within similar return types (e.g., the possibility of multiple detections

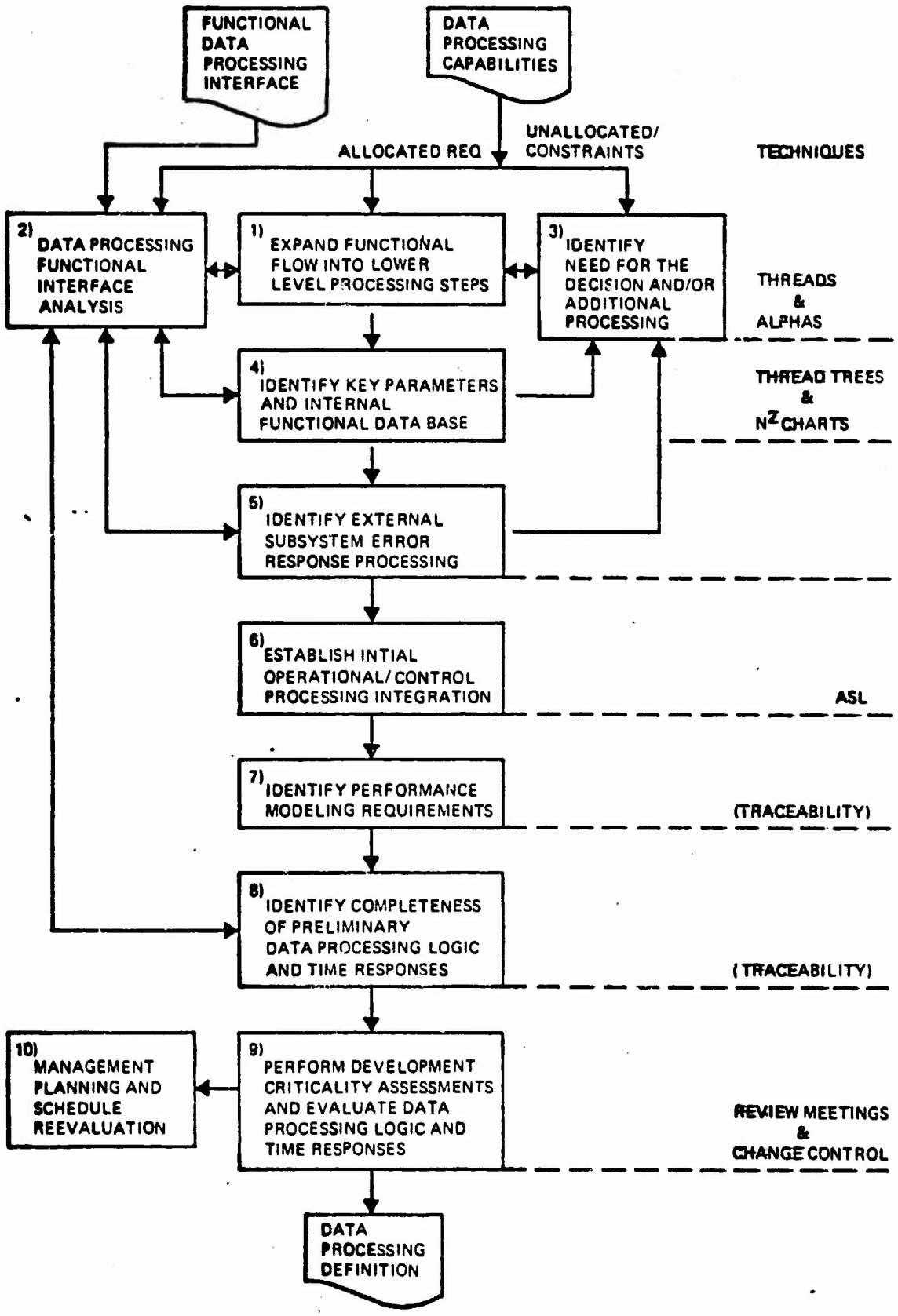


Figure 5-4 ASL and Performance Model Requirements Identification

from a search pulse). This more detailed knowledge of data crossing the interface from the other subsystems to the DP will result in additional threads being identified. In a similar manner, the interface from the DP to other subsystems may require DP outputs not yet accounted for. These required responses can be traced backwards through the DP to determine if the stimulus and condition exist that will result in the required sequence of processing steps. If not, an examination must be made to identify whether the interface definition was incomplete and the stimulus not defined, whether threads need to be added, or whether the response is really not required.

As the functional requirements are expanded and additional threads defined, the requirements for additional processing steps to support internal decision-making are derived. Such requirements result from the fact that many of the conditions for which a desired sequence of processing is to be carried out must first be identified by the DP itself, either from processing of the interface data or processing of historical data saved within the global data base [e.g., the probability that an object is an RV (threatening) must be calculated and tested against some "threshold" or criteria]. Such decision processing steps are derived from the operating rules and the expanded threads, and from the basis for the required DP logic. These decision points are clearly highlighted in the form of thread trees.

Concurrent with the above analyses, the key parameters and the internal functional data base are identified and checked for consistency. Parameters required for decision-making (step 3) are identified and their availability (are they the required output of some processing step?) is verified.

The functional interface definition and external subsystem characteristics must be examined to determine if any additional requirements should be identified for the DP related to errors in the external subsystems. The desired DP responses to bad input data (e.g., garbled radar return message) must be defined either from general error processing requirements in the SRPS or specified by the requirements analyst for subsequent review with system engineering.

To provide a comprehensive overview of the functional requirements and to aid in insuring completeness and consistency of the functional data base, the threads and thread trees are integrated into an ASL. An ASL is a combination

of threads and thread trees with decision points and external interfaces identified. The starting and ending points of an ASL represent input and output ports of the actual DP, hence the external interface of the DP is more clearly described than with individual threads. With completion of a thorough examination of the ASL and resolution of any inconsistencies through iteration (as required) back to previous steps, the functional requirements and logic is defined. At this point the complete (preliminary) high level control structure is defined.

The performance requirements for the processing steps are to be developed during the next phase through trade-off analyses and simulation. To accomplish these analyses, parametric (functional and/or stochastic) performance models must be identified (step 7) and agreed on with systems engineering. This involves design of algorithm sequences and data processor performance requirements which, if met, will satisfy the allocated system performance requirements.

The time response requirements imposed by the SRPS are transferred to the preliminary DP functional definition. Because of the derived requirements (new threads) developed during this phase, some processing threads may not have had time response requirements identified. Such incomplete time requirements will be identified for resolution with system engineering.

The defined DP logic, time response requirements, and parametric model definitions are reviewed with system engineering for completeness and consistency with the SRPS. The management planning and scheduling performed in the first phase is re-evaluated and adjusted based on reassessment of the development criticality.

5.5 ESTABLISH PRELIMINARY PERFORMANCE REQUIREMENTS

In order to establish the preliminary performance requirements, the issues of testability and design freedom must be addressed; and tradeoff analyses must be performed to establish data processor performance. In the previous phase, the individual performance models for data processor capabilities were established; this activity obtains valid boundary conditions such that, if the algorithm design lies within these boundaries, the system as understood will work. This analysis does not say that there is a feasible

software solution nor does it say that a complete set of requirements have been established (these considerations are addressed in later activities). However, this activity is the first to re-allocate the Data Processing performance to individual elements which establishes the requirements on all algorithms and the control structure. A sequence of activities is shown in Figure 5-5.

The primary inputs to this activity are the Data Processing Definition and the Interface information at the element level. This information is analyzed by the construction of a simulator to perform tradeoff and sensitivity studies. The primary focus is the establishment of the boundary condition on each element and on the identification of the initialization and activities of the data base. As these requirements are being definitized, the testing criteria for testability and the degree of design freedom allowable in the software are being established.

After establishing an integrated set of requirements, algorithm boundaries and acceptance criteria can be formulated independent of the future software development. Here the issue of completeness is addressed to ensure that all requirements are accounted for and to determine that this set of requirements is traceable to the set of SRPS requirements from a systems point of view.

Having performed all the activities, a criticality assessment and design evaluation is performed to provide management visibility and to identify potential problems in future activities. At this point in the development cycle, a preliminary set of requirements and their test criteria have been established for the Data Processing Subsystem.

A significant output of this activity is a working, baselined functional simulator which can be used to evaluate the Data Processing Subsystem performance (e.g., MIPS, interface data rate). This can be used to identify subsystem stress points and evaluate subsystem performance sensitivities.

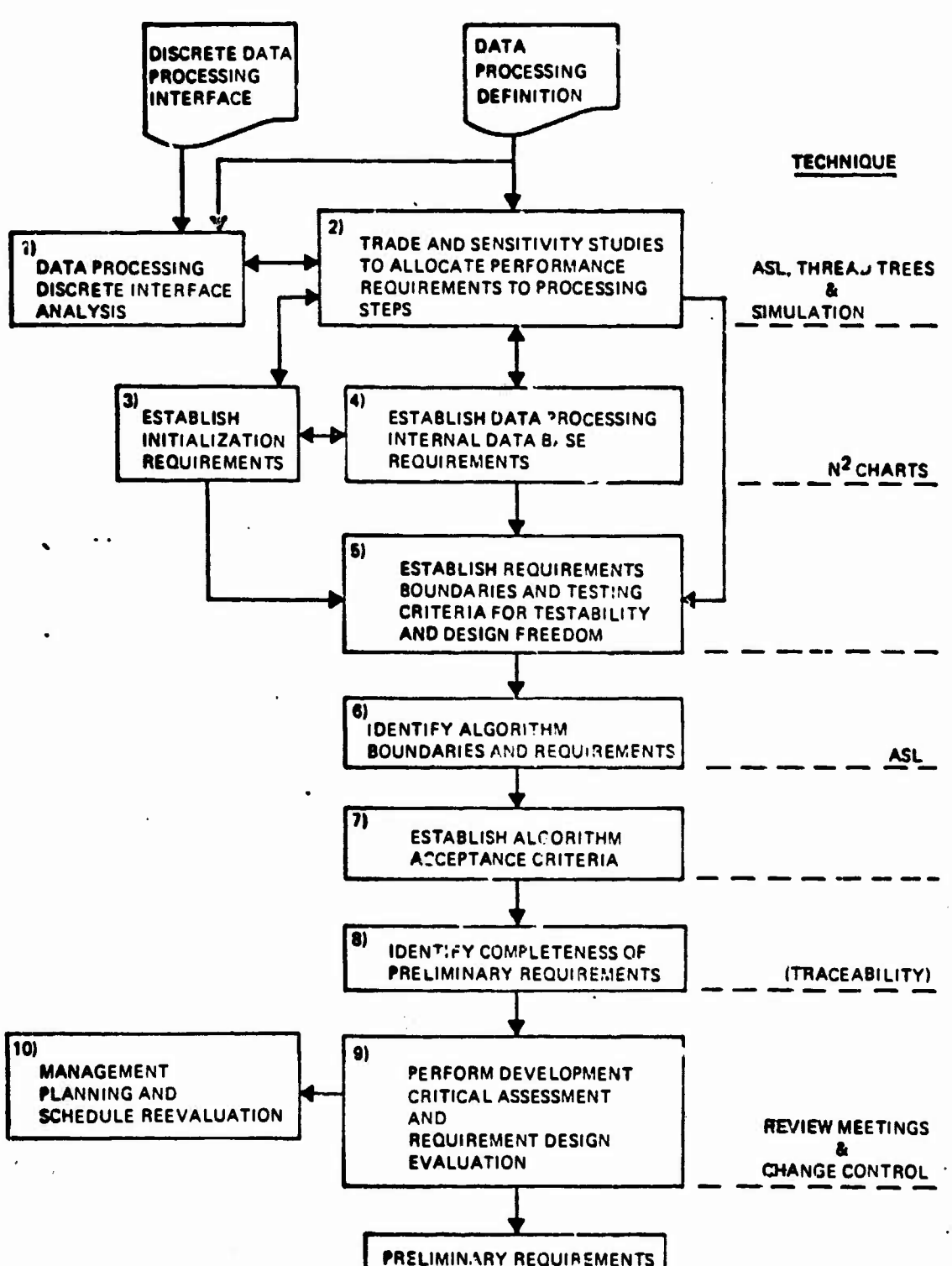


Figure 5-5 Establish Preliminary Performance Requirements

5.6 ALGORITHM DEVELOPMENT

This activity begins with the specification of processing and the set of test criteria. An expertise in numerical analysis is required and, for large, system critical algorithms, competence in the system details and the corresponding physical sciences is necessary. Its result is an assurance of algorithm feasibility.

An outline of the procedural actions in algorithm development is depicted in Figure 5-6. The first step is to define and examine algorithm input/output requirements and determine that sufficient information is available to proceed in the analysis. Then a survey of the state-of-the-art is used to determine various approaches to the solution. It is desirable to acquire a multitude of acceptable algorithmic approaches. The resulting algorithm preliminary design serves two functions: On the one hand, it leads to a set of necessary engineering assumptions which needs to be clearly documented (e.g., characteristics of specific radar waveform returns); and on the other hand, it establishes a particular procedure for satisfying the specification. Following the algorithm design, its design feasibility and computational characteristics are examined in detail. An algorithm having satisfied all requirements up to this point is designed and coded. Next, the algorithm is unit tested via some test driver. Finally, the test results and design are reviewed in order to establish that the specification and test criteria are satisfied.

The key outputs of this analysis are a documentation of the assumed characteristics of the physical and processing environment, and numerical procedures which can perform the desired processing.

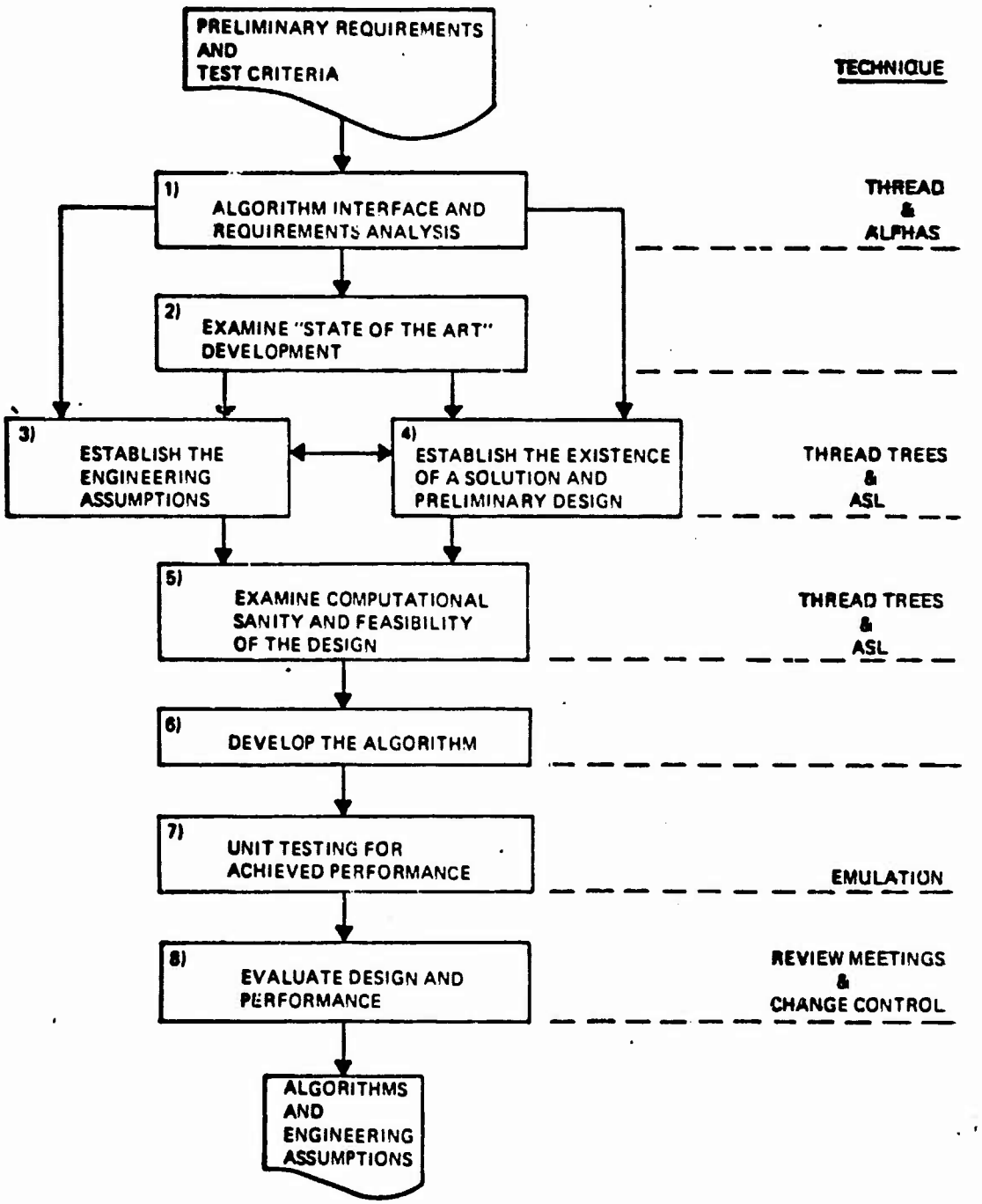


Figure 5-6 Algorithm Development

5.7 ALGORITHM/ASL INTEGRATION

The algorithm/ASL integration phase of SREM involves two somewhat distinct and separable activities: 1) algorithm integration and 2) test procedure generation, as shown in Figure 5-7. Algorithm integration addresses the incorporation of the algorithm logic design into a simulation framework which assures that all processing paths and test points described in the Preliminary Performance Requirements (PPR) exist. Test procedure generation is that activity which generates detailed scenarios to test the algorithms in an integrated environment.

The process of integrating the algorithms involves not only the actual integration but the design and development of a non-real-time version of the software (the emulator) and its preliminary testing. The actual integration must be performed such that the algorithms are mapped into the structured requirements format. In the process, algorithm nomenclature is made consistent (if necessary), and algorithm assumptions, I/O interfaces, limitations, and logic are made consistent and compatible. If the algorithm is designated for implementation in the real-time code, evaluations of analytical feasibility must be performed. For example, an algorithm which does not converge under certain circumstances must be isolated and either replaced or appropriate ranges of inputs specified and controlled.

The integration can be accomplished in successive stages. First, neighboring algorithms in threads can be integrated, until a whole tree of threads can process input data to completion. This assures that the primary interactions are consistent; the second level interactions are checked for consistency in the emulator integration.

Subsequent to the initial integration procedures, the non-real-time process would be implemented. All logic and transformations representing the tactical software processing (not computer scheduling and operating system) would be transformed to a tactical software emulator. This program when exercised with a high fidelity SETS, constitutes an existence proof: Algorithm compatibility and system completeness are demonstrated. Requirements completeness is demonstrated by re-testing each algorithm in the emulator in the stand-alone mode using algorithm tests derived in prior phases and by system tests appropriately defined for that purpose (i.e., scenarios which exercise all possible routes and provide off-nominal inputs).

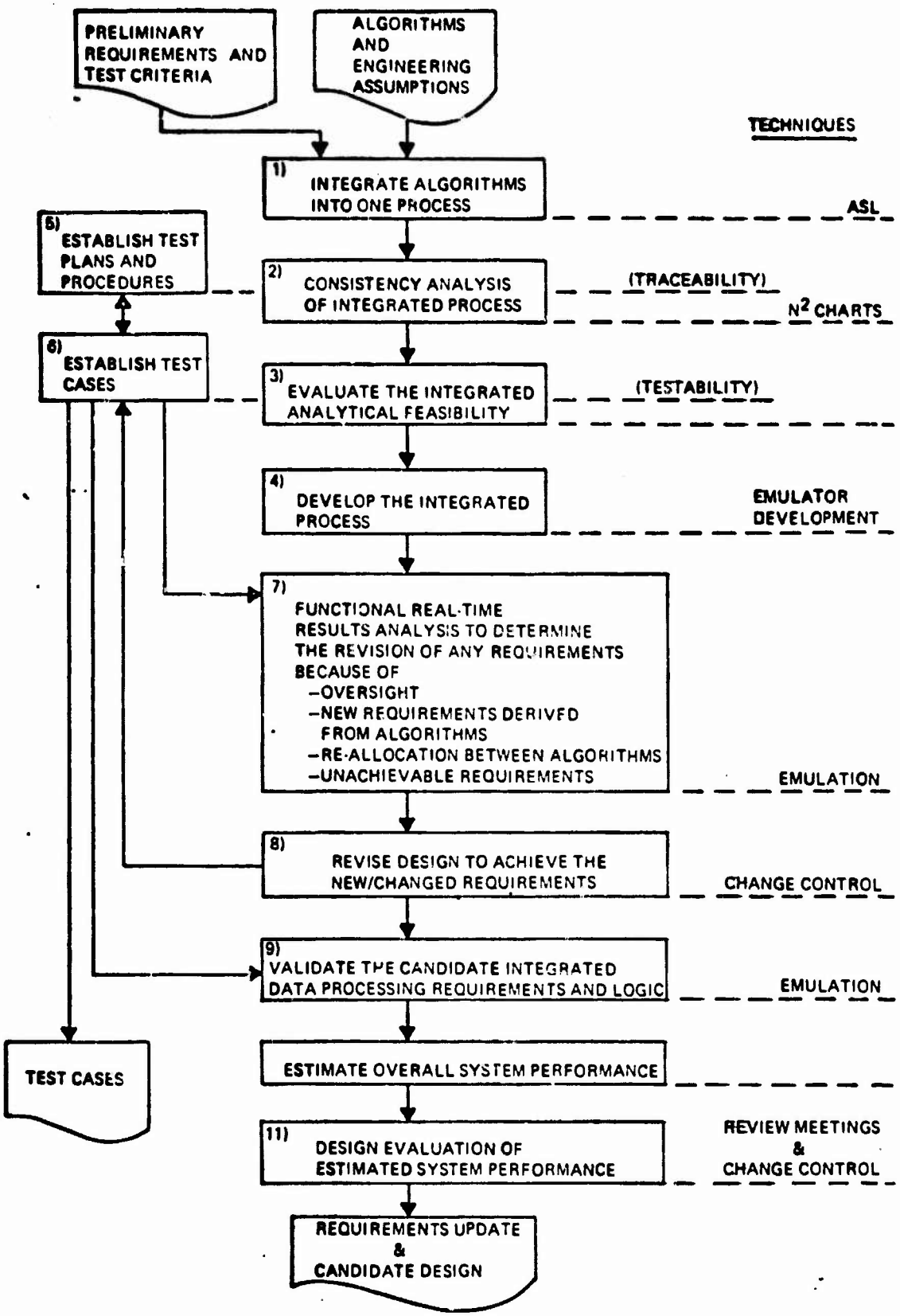


Figure 5-7 Algorithm/ASL Integration

The design of an emulator includes the design of a software structure, data base structure and access/update techniques, scheduling techniques, and techniques to interface with SETS. It has many of the problems of the real-time software design, including run-time and storage limitations.

All of the above activities will identify inconsistencies and compatibility problems which must be continually worked by algorithm and requirements modifications. Furthermore, analysis of the results of the testing will determine if revisions to requirements are necessary because of omission due to oversight, newly derived requirements from algorithms, re-allocation of requirements between algorithms, or unachievable performance requirements.

Test procedure generation involves transforming the preliminary performance requirements and test criterion into detailed scenarios for the aforementioned purpose. Specifically, the scenarios must be designed to demonstrate the completeness and consistency of specified requirements, demonstrate off-nominal conditions and degraded performance. They must be designed as RTSW tests as well as Emulator Tests. In addition to scenario design, the test procedures must specify what data is to be collected, where probes are to be placed, and display formats such that algorithm and DP system performance can be collected and analyzed. The test procedures should specify additional data to be collected by the emulator which indicates the state of the system. For example, data should be collected on threatening object arrival rates, queue lengths, etc. in the validation phase to provide operating characteristics to the process designer. It should be noted that the full range of tests (i.e., load tests) may not be exercised using the emulator during validation. Because of the complexity of an emulator (i.e., non-real-time tactical software and SETS software) and the limitations of the computer hardware used for emulator development, it may be feasible to test completeness, consistency, and off-nominal performance and not feasible to perform load tests. In this case, the algorithm performance data collected in the emulator tests could be used to calibrate the algorithm models in the functional simulator which is used to gather system performance under load.

The significant outputs of the algorithm/ASL integration activities are: A working, baselined data processing emulator which can be used to assure the consistency of the requirements, to perform tradeoff and sensitivity analyses, and to validate the requirements; and a set of test cases which ensure that the requirements are testable, and an integrated set of algorithms and an ASL.

5.8 PERFORMANCE ANALYSIS AND EVALUATION

The purpose of this final step is to check for completeness of the requirements by detailed simulation. At the start of this step, the previous analysis has done everything possible to assure completeness and an estimation of the total subsystem performance has been made. The hypothesis is if that estimate is proven correct by test cases, one has heightened confidence in prediction of performance for cases not run, and for alternative design solutions. Therefore, to assure validity one would like to determine that the estimated performance and the inherent engineering assumptions are correct, such that, when all subsystems are integrated, system performance is obtained. Figure 5-8 shows a sequence of activities which this analysis is to accomplish.

The task of performance analysis and evaluation is very seldom done in large programs. This is because previous tasks sometimes overrun in time and there is insufficient time to carry out this analysis; or because insufficient traceability and visibility exist to establish the type of testing to be carried out. The question of how much testing is required to achieve a high degree of completeness and validity is another problem.

The first steps are to design test cases for the predicted stressing points of the Data Processing Subsystem and to exercise or verify the engineering design assumption. Having performed detailed simulation, the next step is to evaluate the data processing achieved performance given that the other subsystems are fixed.

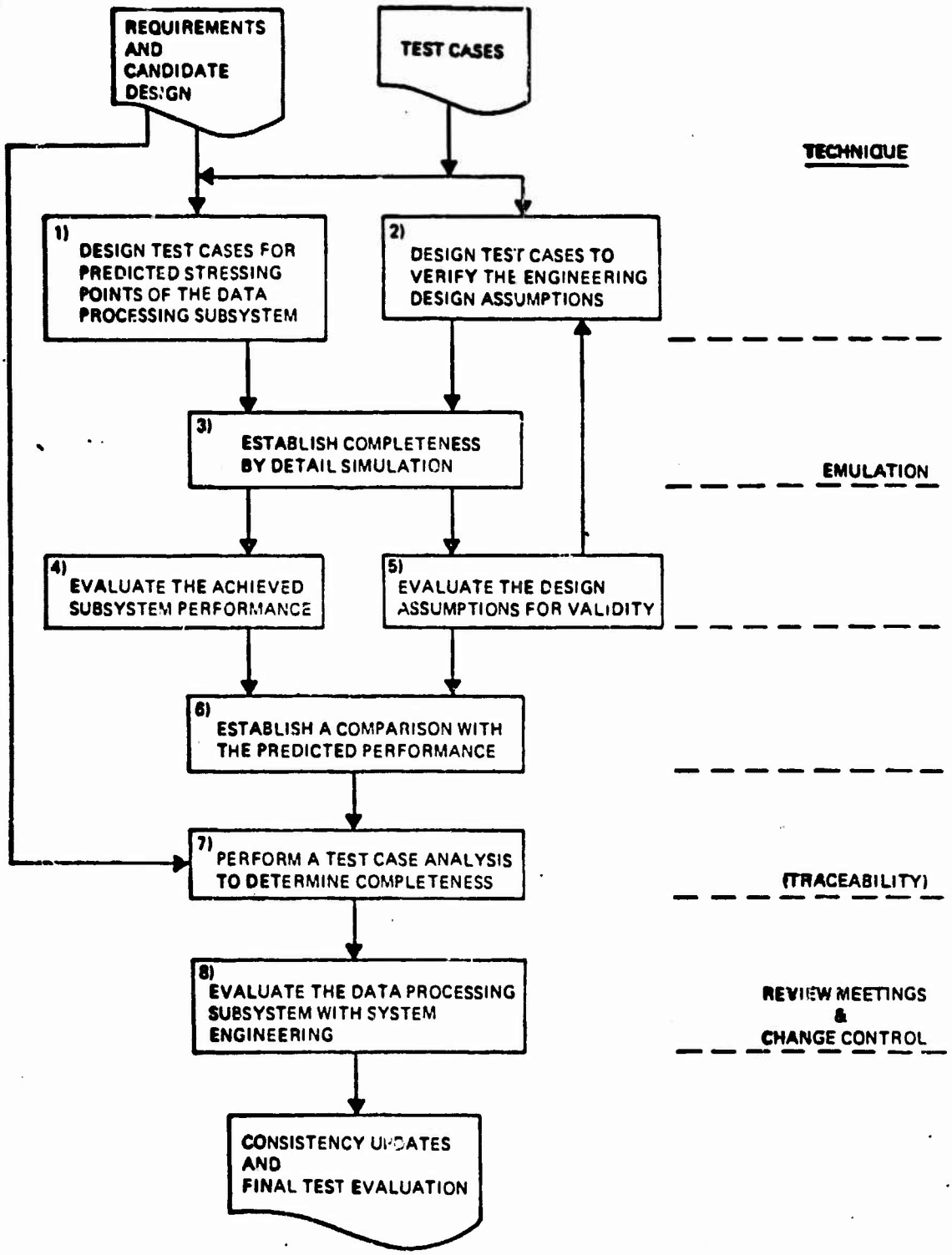


Figure 5-8 Performance Analysis and Evaluation

Next, the achieved performance is compared to the estimated performance to determine if the Data Processing Subsystem is better-than-expected or worse-than-expected. If this comparison is incomplete, the establishment of additional test cases may be required. The prime objective is to establish a complete set of test cases which exercise all requirements for the Data Processing subsystem.

The final products of this activity are the consistency updates and a complete test evaluation of the Data Processing Subsystem.

5.9 METHODOLOGY CAPABILITIES SUMMARY

The previous subsections presented the description of the identified steps necessary to produce a validated software specification from a SRPS. The expanded descriptions of each step (Sections 5.3-5.8) identify the capabilities required, or implied, of the final integrated methodology. These capabilities must be achieved through the engineering techniques, support tools and language, and the detailed procedures which define the integrated methodology under development. The required capabilities are summarized in Table 5.1.

Table 5.1 Summary of Required Capabilities

CAPABILITY TO...	MAJOR STEPS* 1 SRPS REQS. ANAL.	2 ASL & PERF. MODEL REQS.	3 PREL. PERF. REQS.	4 ALGORITHM DEV.	5 ALGORITHM/ ASL INTEG.	6 PERF. ANAL. & E'VAL.
Identify functional requirements and enumerate for traceability	X					
Classify functional requirements into identifiable categories	X					
Identify and enumerate for traceability interface requirements and interface data definitions at both functional and discrete levels	X	X	X	X		
Associate DP input interface data with DP output interface data	X	X	X	X		
Associate functional requirements with DP interface requirements and data definitions via interface to interface sequences (threads)	X	X	X	X		
Identify and enumerate performance requirements, and associate performance requirements with functional requirements and/or functional categories	X					
Expand identified functional performance requirements into one or more lower level (more detailed processing steps with traceable identifiers)		X				
Define and enumerate for traceability derived functional processing steps (transformation or decision) and associate them with processing sequences (threads)		X				

Table 5.1 Summary of Required Capabilities (Continued)

CAPABILITY TO...	MAJOR STEPS*: 1 SRPS REQS. ANAL.	2 ASL & PERF. MODEL REQS.	3 PREL. PERF. REQS.	4 ALGORITHM DEV.	5 ALGORITHM/ ASL INTEG.	6 PERF. ANAL. & EVAL.
Check the logical consistency and completeness of requirements at each level of development through static testing, examination of input/output compatibilities and identification of redundant, conflicting or unneeded processing steps	X	X	X		X	X
Assess development criticality and associate criticality measures to traceable elements (e.g., processing steps, threads, functional categories, algorithms)	X	X	X			
Plan, schedule and assess status of subsequent requirements engineering activities in terms of traceable elements and modify plans based on results and criticality assessment	X	X	X			
Define requirements on performance models and identify key system parameters for development of functional and/or stochastic simulation, and associate with processing steps, processing sequences, or functional categories		X				
Identify and enumerate for traceability external subsystem data error modes and/or failure modes and associate with derived DP requirements		X				
Synthesize processing sequences into integrated sets (e.g., ASL or trees) based on decision processing steps, common processing steps and common internal or external data interfaces		X	X		X	

Table 5.1 Summary of Required Capabilities (Continued)

CAPABILITY TO...	MAJOR STEPS* 1 SRPS REQS. ANAL.	2 ASL & PERF. MODEL REQS.	3 PREL. PERF. REQS.	4 ALGORITHM DEV.	5 ALGORITHM/ ASL INTEG.	6 PERF. ANAL. & EVAL.
Assign and associate data characteristic (e.g., origin, destination, range, units, etc.) to both functional and discrete data sets and data elements to maintain internal system data base and identify initialization requirements			X		X	
Extract and modify performance measures to perform tradeoff and sensitivity studies			X		X	X
Establish performance requirements tolerances and associated test or acceptance criteria for design freedom			X		X	
Define algorithm boundaries and acceptance criteria and maintain association between algorithm boundaries and processing steps, processing sequences or integrated sub-sets for future integrations and maintain previously defined algorithms			X		X	
Maintain record of important engineering assumptions and associate with affected algorithms			X	X		
Assess computational sanity, efficiency, and implementation possibility in computer-independent terms		X	X	X	X	
Develop and save test plans, procedures and cases (data) at multi-levels (e.g., algorithm, processing sequences, DP subsystem)			X	X	X	X
Save and evaluate test results, and compare to previous results			X	X	X	X

Table 5.1 Summary of Required Capabilities (Continued)

CAPABILITY TO...	MAJOR STEPS* 1 SRPS REQS. ANAL.	2 ASL & PERF. MODEL REQS.	3 PREL. PERF. REQS.	4 ALGORITHM DEV.	5 ALGORITHM/ ASL INTEG.	6 PERF. ANAL. & EVAL.
Estimate and extrapolate system performance for various conditions from the specified requirements and simulation results for discrete sets of conditions		X	X		X	X
Assign information source identifiers to all elements for configuration control and traceability	X	X	X	X		
Maintain an unambiguous correlation between a set of requirements and the simulator being used to validate them		X	X	X	X	X
Produce specifications and engineering documentation (e.g., test results) from the same data base and requirements set that was used for simulation and analysis	X	X	X	X	X	X
Control access to and modification of the information base for security considerations and to preclude unauthorized changes	X	X	X	X	X	X

*MAJOR STEPS

1. SRPS Requirements Analysis
2. ASL & Performance Model Requirements Identification
3. Establish Preliminary Performance Requirements
4. Algorithm Development
5. Algorithm/ASL Integration
6. Performance Analysis and Evaluation

6.0 GLOSSARY

ASL	Algorithm Sequencing Logic
BMD	Ballistic Missile Defense
BMDATC	Ballistic Missile Defense Advanced Technology Center
DP	Data Processor
Emulator	Non-real-time implementation of tactical software. Incorporates actual transfer functions but operates in a batch environment with a SETS.
MIPS	Million instructions per second
N^2 Chart	Matrix depicting I/O interfaces (also known as diagonal chart)
PPR	Process Performance Requirements
SETS	System Environment and Threat Simulation
SREP	Software Requirements Engineering Program
SRPS	System Requirements and Performance Specification
Static Testing	That portion of software testing which can be accomplished prior to execution of that software, e.g., I/O consistency between elements of thread.
Thread	The sequence of processing steps within a subsystem which is initiated by a stimulus at an input port and results in the appropriate response at an output port or internal process termination point.
Tree	A tree is a directed graph that contains no loop and at most one branch entering each node, and shows processing sequences in an integrated form.

7.0 REFERENCES

1. "ASSD Software Performance Requirements - Software Requirements Engineering Methodology," TRW Report No. 22944-6921-011, August 15, 1974 (CDRL G009, DAHC60-71-C-0049).

2. "ASSD Software Performance Requirements - Software Requirements Language," TRW Report No. 22944-6921-016, October 1, 1974 (CDRL G00G, DAHC60-71-C-0049).

3. "ABMDA Research and Development Software Standards," SDC, September 8, 1972.

4. "ASSD Special Report - Current Software Requirements Engineering Technology," TRW Report No. 22944-6921-010 (CDRL G00A, DAHC60-71-C-0049).

5. "ASSD Test and Evaluation Procedure - Software Requirements Engineering Methodology," TRW Report No. 22944-6921-013, September 15, 1974 (CDRL G00F, DAHC60-71-C-0049).

22944 - 6921 - 013

TEST AND EVALUATION PROCEDURE - SOFTWARE REQUIREMENTS ENGINEERING METHODOLOGY

CDRL GOOF

SEPTEMBER 15, 1974

Sponsored By
BALLISTIC MISSILE DEFENSE
ADVANCED TECHNOLOGY CENTER

DAHC60-71-C-0049

TRW
SYSTEMS GROUP
Huntsville, Alabama

TEST AND EVALUATION PROCEDURE - SOFTWARE REQUIREMENTS ENGINEERING METHODOLOGY

CDRL GOOF

SEPTEMBER 15, 1974

Sponsored By
Ballistic Missile Defense
Advanced Technology Center
DAHC60-71-C-0049

TRW
SYSTEMS GROUP
Huntsville, Alabama

TEST AND EVALUATION PROCEDURE - SOFTWARE REQUIREMENTS ENGINEERING METHODOLOGY

CDRL GOOF

SEPTEMBER 15, 1974

Principal Author

M. W. Alford

Approved By

T. W. Kampe, Manager
Software Requirements
Engineering Program

for R R Boyer
James E. Long, Manager
Huntsville Army Support Facility

Sponsored By
Ballistic Missile Defense
Advanced Technology Center
DAHC60-71-C-0049

TRW
SYSTEMS GROUP
Huntsville, Alabama

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

1.0 OVERVIEW

This report identifies the tests and experiments to be performed using the Software Requirements Engineering Methodology, their schedules, and the procedures to be used to evaluate their success. Section 2.0 provides the rationale for the design of the tests and experiments. The interactions and objectives of each experiment and test are summarized and an approximate schedule is provided. Section 3.0 summarizes the procedures for carrying out and documenting the results of the experiments. Section 4.0 provides a more detailed description of each experiment and test.

In this context, an experiment is the name given to the application of a set of techniques to a given problem; a test is the name given to the application of a specific set of tools and procedures to a given problem, where the tools and procedures have been previously validated. In the case of a test, specific quantitative measures of success may be one of the objectives of the test.

1.1 PURPOSE OF THE EXPERIMENTS

The purposes of performing experiments of a Software Requirements Engineering Methodology are more complicated than just to assure that the methodology applies to a given class of problem. The experiments are being conducted in part to improve the methodology and to build a data base for definitizing its implementation.

A methodology can be conceived of having four parts:

1) A set of ideas which are to be exploited in dealing with problems of a particular type;

2) A set of techniques which apply to those ideas to specific problems;

3) A set of tools which assist in the implementation of the techniques; and

4) A set of procedures which describe when and how to use the tools and techniques, describing both the conditions under which they are to be used and how to employ them to get the desired results.

The ideas of the Software Requirements Engineering Methodology have been described in [2] and are to be further refined in future documentation. The techniques are to be described in the Methodology Capabilities Report [3]. The tools and procedures are to be described next year.

At this stage of the methodology's development, the purposes of the experiments can therefore be classified in the following ways:

- Identify any holes in the methodology, so that they can be fixed. At a fundamental level, the ideas and preliminary set of techniques must be validated for completeness.
- Provide a preliminary, qualitative assessment of the methodology.
- Exercise the preliminary techniques in order to evaluate the benefits of proposed tools.
- Prepare a set of examples in order to prepare and evaluate the requirements for a set of tools to implement the techniques.
- Gather a set of examples to be used to communicate the methodology to others.
- Gather the data base by which quantitative "measures of success" of the methodology might be assessed.

The overall set of experiments and tests described in Section 2.2 were selected in order to assure all of the above objectives without an inadvertent duplication of effort. Figure 1-1 provides a cross-reference between the experiments and these goals. Each of these objectives is a necessary one in order to be assured that the final methodology is complete, implements the most cost effective tools, and has a sufficient set of documented examples that it can be communicated effectively to others who have not taken part in the research effort. Each experiment or test contributes to more than one objective.

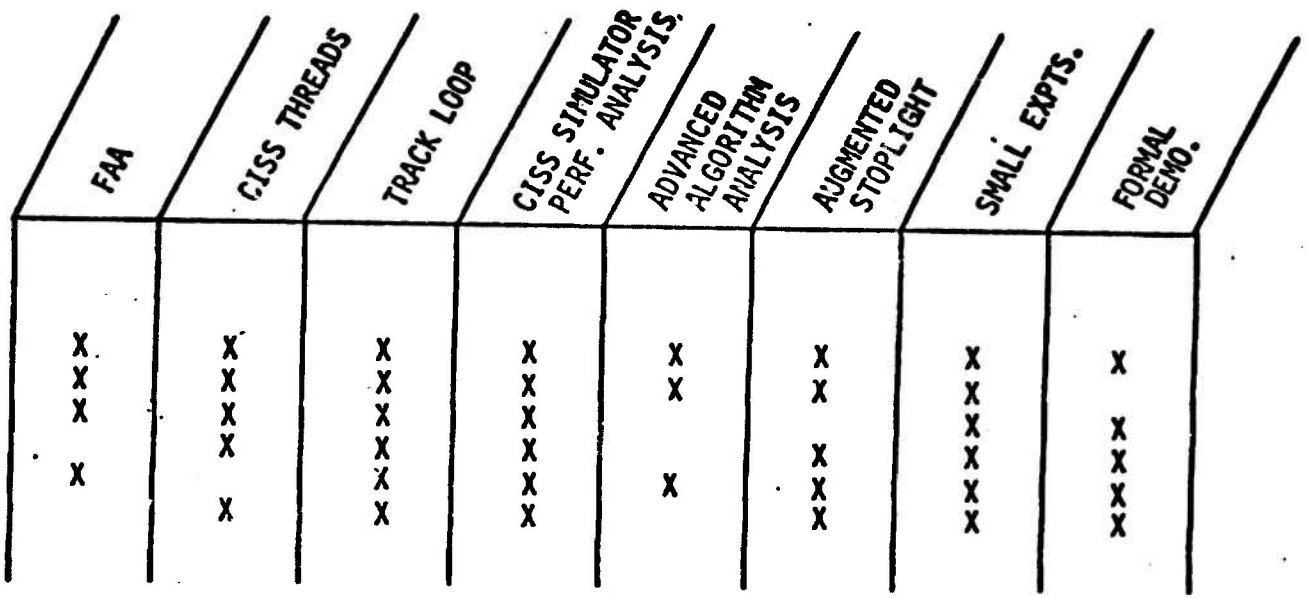


Figure 1-1 Experiment Objectives Checklist

1.2 RELATIONSHIP OF THE EXPERIMENTS TO THE METHODOLOGY DEMONSTRATION

It is useful to consider the manner in which the Chief Programmer Team methodology was developed and tested before it was published and proposed as an industry standard by IBM. First, H. Mills had an idea which promised that a new blending of the concepts of structured programming, top-down programming, and an organizational concept would produce a better way to produce computer programs. Mills published these ideas within IBM and convinced management to carry out an experiment to demonstrate the validity and applicability of these ideas. Some preliminary procedures were developed, and then they were implemented on an experimental basis. Some "measures of success" were identified and computed (e.g., cost per instruction, reliability of the resulting code, the qualitative assessment of the documentation). His procedures and tools were modified to take this experimental data into account, and the methodology was selected for use in the New York Times job [4,5]. Data was collected from which the measures of success could be obtained, and the results published. Note that, whereas Mills repeated a job which someone else had done in order to obtain comparative data for experimental purposes, the methodology demonstration (N. Y. Times job) did not duplicate a previous effort. This is the general approach followed to experiment and test the Software Requirements Engineering Methodology. This also points out the relationships of the experiments (used to justify the expense of a demonstration) and of the methodology demonstration (used to demonstrate the utility of the methodology).

2.0 EXPERIMENT OVERVIEW

2.1 SELECTION CONSTRAINTS

The selection of the experiments and tests of the methodology must satisfy a collection of quite diverse constraints, the first of which is the available effort which can be devoted to conduct and analyze them. Other such constraints are:

1) Many small experiments which produce more timely results are to be preferred to one larger experiment for initial experimentation.

2) The range of systems considered should include the definition of a new system, the description of an existing software specification using the methodology, and the modification of a software subsystem which is described by a software specification.

3) The experiments shall include interactions with GRC and TI for the definition of a SRPS and a software specification in order to achieve some degree of involvement and familiarity with the Software Requirements Engineering Program.

4) At least one non-BMD problem should be included in the experiments; but for the most part, the remainder of the experiments should be oriented towards BMD-like systems.

5) At least one full-scale test which covers the range of SRPS to software specification should be planned, and one preliminary experiment of a smaller magnitude to assure the continuity of the methodology. Since the methodology is to address the problem of complexity of software specifications, the test (or demonstration) will address a fairly large software development problem.

6) The experiments should be scheduled in time to focus on the more critical problems first.

A checklist of the experiments discussed in Section 2.2 against the above constraints appears in Figure 2-1.

The experiments shall totally cover the range of methodology activities from the definition of the SRPS to the interaction with the process designer after receipt of a software specification. These activities can loosely be divided into:

CONSTRAINTS	FAA	CISS THREADS	TRACK LOOP	CISS SIMULATOR PERF. ANALYSIS	AUVANCED ALGORITHM ANALYSIS	AUGMENTED STOPLIGHT	SMALL EXPTS.	FORMAL DEMO.
1. SMALL EXPTS	X		X	X	X	X	X	
2. DEFINE NEW SYSTEM	X					X		X
DESCRIBE EXISTING SYSTEM		X	X	X			X	
MODIFY SYSTEM					X			
3. INTERACT WITH GRC	X		X		X	X		X
INTERACT WITH TI			X	X	X	X		X
4. NON-BMD PROBLEM	X					X	X	
BMD PROBLEM		X	X	X	X		X	X
5. FULL RANGE OF ACTIVITIES			X			X		X
6. MOST CRITICAL FIRST.	X	X		X	X			

Figure 2-1 Experiment Constraints Checklist

22944-6921-013

- SRPS specification
- SRPS to threads transition
- Refinement of threads
- Definition of Software Performance Requirements
- Definition of a Process Performance Requirements Specification (PPR)
- Interaction with Process Design.

The SRPS definition includes the writing of a SRPS-like document for the problem at hand, with possible interactions between the SRPS designer and the software requirements analyst. The critical questions to be answered are: How do you start? and what sort of interactions are required?

The refinement of threads consists of those activities necessary to refine the definition of the software specifications down to the equation level. The key questions to be addressed center about the level of detail necessary for a sufficient specification, the determination of completeness and consistency of the specification, and the traceability of the specification back to the SRPS content.

The performance requirements activities include the building of simulators to validate the specification, and their use to obtain performance indicators and requirements such as accuracies and time response requirements. Issues to be examined are the degree of automatability of the simulator building and exercise processes, simulator documentation and validation, and the ability to extract the performance requirements of the software.

The Process Performance Requirements definition includes the activities of writing the software specification. Issues to be examined are specifically those of traceability, ease of documentation, management visibility of the progress of the specification, and the validation and testability of the requirements themselves.

The relationship of the experiments selected to the range of methodology activities is provided in Figure 2-2.

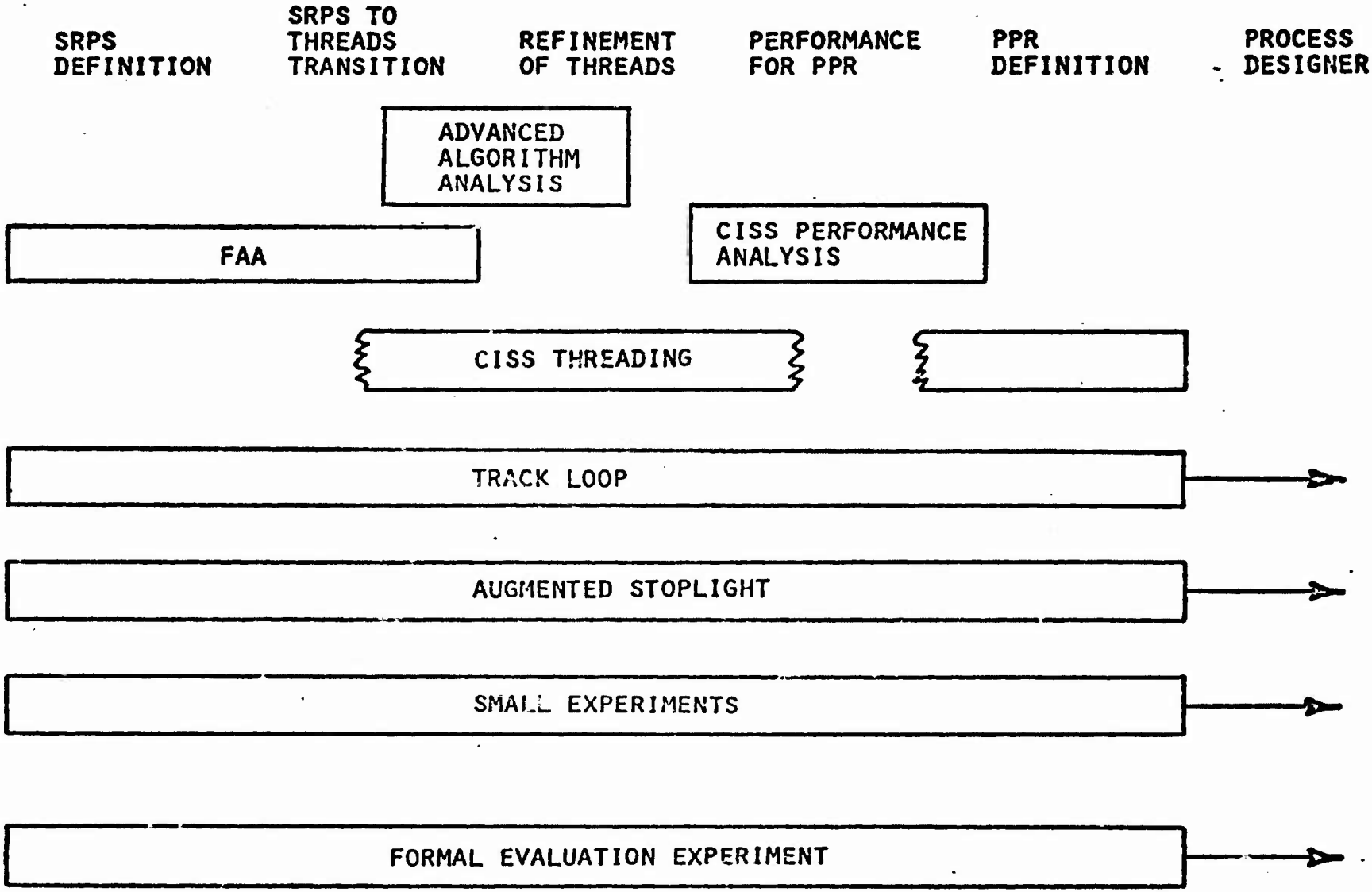


Figure 2-2 Methodology Synthesis Experimentation

2.2 SELECTION RATIONALE

The experiments selected to satisfy the aforementioned objectives and constraints are:

- A top-level description of the FAA Air Traffic Control System
- The Threading of the TDP CISS
- The TDP Track Loop Performance Analysis
- The CISS Simulator Performance Analysis
- Advanced BMD Algorithm Evaluation
- Augmented Stoplight Problem
- Additional Small Experiments (to be specified later)
- The Formal Evaluation Experiment (based on an Algorithm Test Bed concept, to be specified in 1975).

These experiments will be individually described in Section 4.0. Figure 2-3 presents an approximate schedule for these experiments. Recall that Figure 1-1 presents a checklist of the experiments to the experiment goals in Section 1.0, and Figure 2-1 to the experiment constraints in Section 2.0; Figure 2-2 presented the range of methodology activities to be addressed.

The FAA Experiment was chosen as the non-BMD experiment and to address the problems of "what to do first;" the depth to which the analysis will go is not envisioned to be very far into the refinement of the threads.

The CISS Threading Experiment will address the questions of the number of threads in BMD software. This will use the previously developed TDP CISS and will focus on problems of representation of the processing requirements in a threaded format. The results of the analysis will be provided to TI for comment, which will be included in the evaluation.

The TDP Track Loop Performance Analysis is to be performed for inclusion into the documentation which describes the format and content of the SRPS and CISS, and how to get from one to the other. The results are to be provided to TI and GRC for comment and analysis.

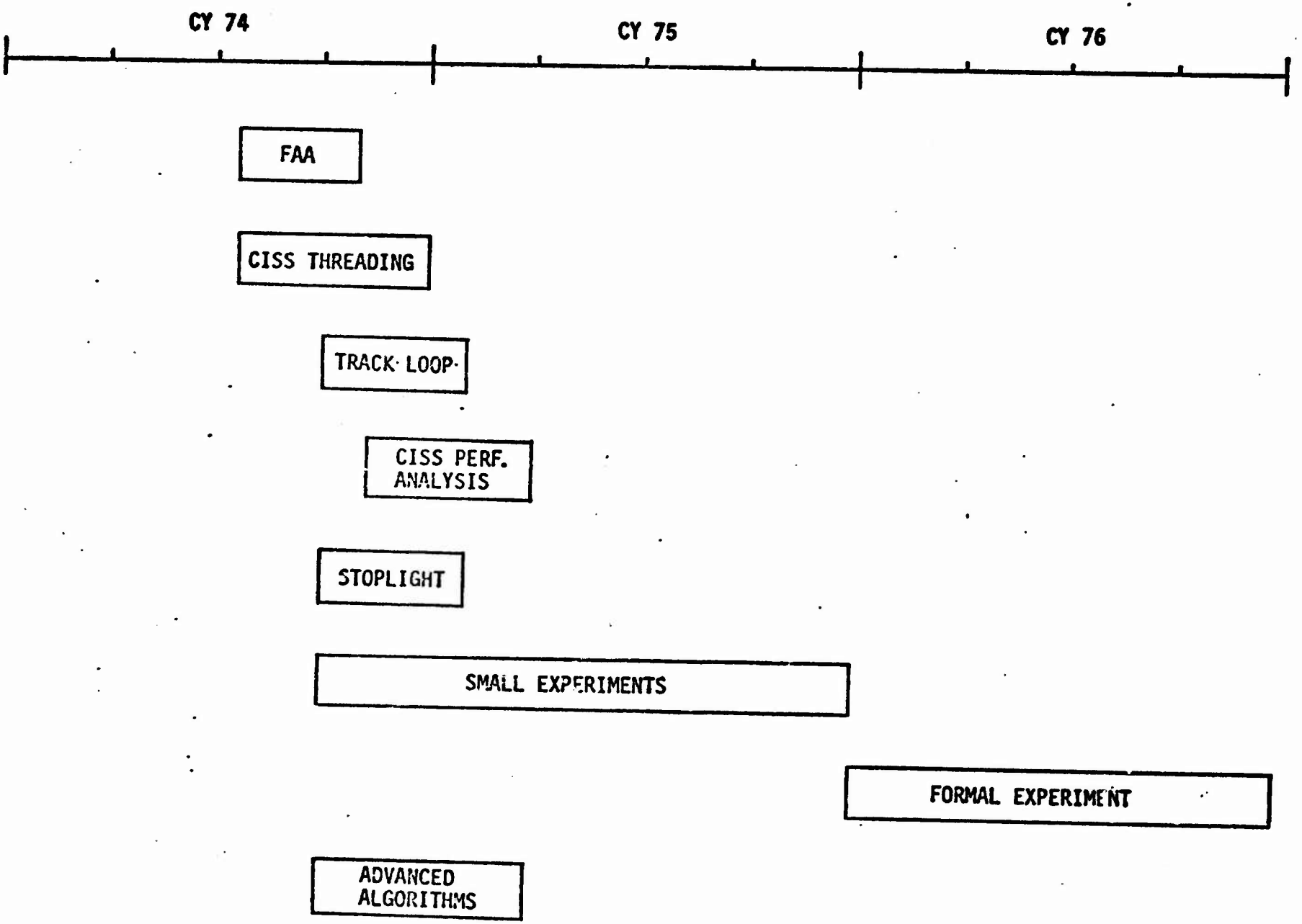


Figure 2-3 Experiment Schedules

The CISS Simulator Performance Analysis Experiment is actually a series of interrelated analysis efforts to determine the processing performance requirements which can be extracted from the CISS Simulator. The current CISS describes the processing to be performed but does not include quantitative performance requirements on computational accuracy and on time response requirements. The purpose of the experiment is to fill this gap and thereby provide examples of such analysis.

The Advanced BMD Algorithm Evaluation Experiment is another series of analysis efforts to do preliminary evaluation of proposed system-level algorithm definitions. This will specifically address a commonly asked question of: Given "off-the-shelf" algorithms, how does one proceed to integrate them into a software system already described by a specification?

The Augmented Stoplight Experiment is a re-do of the previous Stoplight problem; the previous description is being augmented to make it more realistic. This experiment will cover the range of activities and will specifically address the critical problem of automatic simulator construction. The SRPS written for this problem will be given to GRC for comment and the resulting software specification will be given to TI for comment.

An additional set of small experiments will probably take place which are currently not scheduled. The extent and number of these experiments will depend partially on available schedule and funding available. This will include the "case studies" which analyze existing projects.

The Formal Evaluation Experiment is presently envisioned to be some variant of an Algorithm Test Bed and will be a multi-year experiment in which GRC writes the SRPS and perhaps TI produces the software. A separate Test and Evaluation Procedure will be written for this experiment in which quantitative measures of success and measurement techniques to collect necessary data will be called out. This is the formal methodology validation test to be used to demonstrate the utility of the Software Requirements Engineering Methodology.

3.0 EXPERIMENTAL PROCEDURES

The procedure for performing an experiment to meet the objectives of Sections 1.0 and 2.0 are rather straightforward. One defines the objectives of the specific experiment and identifies the nature of the example to which the methodology ideas and techniques are to be applied. The results of the analysis are documented as an example of the application of the methodology, without any discussion of the effectiveness or applicability of the techniques used. Finally, the experiment itself is subjected to a critique, conclusions are drawn and documented concerning the effectiveness of the methodology or its documentation, and recommendations are made as appropriate concerning the utility of proposed tools and/or augmentations to the methodology.

The results of the experiments are to be maintained in a loose-leaf notebook as the experiment is in progress (much like the Unit Development Folders for the software development). Thus, each experiment will be completely documented in one informal report which will be reproduced and distributed periodically until the end of the contract reporting period, at which time the collection of reports will form the basis for a Test and Evaluation Report on the Methodology.

The contents of this informal report will include, but not necessarily be limited to, the following items:

1) A statement of the objectives of the experiment. This will reference the objectives of the experiments described in the previous section.

2) The conditions of the experiment. This shall include the schedule and manpower allocated, the documentation available which describes the example to which the methodology is to be applied, etc.

3) A summary of the experimental results. This may include: A statement of what happened during the experiment (e.g., including the number of false starts, if any); a description of how and the order in which the methodology techniques were applied to the example; and the results of the analysis. This will be, in short, the contents of a good experiment notebook. Appendices may be referenced in order to organize the bulk of the material. This material should not, however, contain conclusions or recommendations of any sort; it is to be an objective summary of the course of the experiment.

4) A summary of conclusions of the experiment itself. This set of conclusions addresses the objectives of the experiment, and should be limited in scope.

5) A critique of the experiment. This section may draw conclusions concerning the effectivity of the methodology, may make recommendations concerning the methodology techniques, tools, and proposed procedures, and may critique the way in which the experiment itself was conducted. This section addresses such questions as:

- Was the methodology sufficient to handle this example?
- Are there difficulties with the documentation of the methodology?
- Do any quantifiable "measures of success" suggest themselves?
- If the example involves a SRPS, does GRC agree with the level of detail and interactions between system designer and software analyst implied by this example?
- If the example involves a software specification, does TI agree with the level of detail and amount of interactions implied?
- If an alternate form of the specification is available, are the software requirements clearer, or more understandable, or less constraining than the previous form?
- Were there difficulties expressing certain types of requirements in the intended format?
- Does the example suggest a preliminary evaluation of the techniques and desirable degree of automation of proposed tools?
- Is the example as documented suitable to communicate some aspect of the methodology?

The conditions of the experiment will include a proposed schedule for the documentation of the experiment. If these schedules change, they will be updated in the document.

4.0 EXPERIMENT DESCRIPTION

The specific experiments identified in the previous section are described below. This material constitutes the preliminary information to be included in the experiment notebook concerning the objectives and constraints of the experiments.

4.1 FAA AIR TRAFFIC CONTROL EXPERIMENT

In December, 1973, TRW published the results of a study of automation applications for an Advanced Air Traffic Management System. This work was performed for the Department of Transportation/Transportation Systems Center (DOT/TSC), under Contract Number DOT-TSC-512. The reports describe a generic Air Traffic Control Management System, in which no man/machine allocations have been made, in terms of a function to be performed.

The experiment will start with the derivation of the top-level software requirements of the Function 2 (Control Traffic Flow); if resources permit, the scope of the experiment may later be extended to the remainder of the system. The primary focus is to answer the question: What does one do first to start the process? Is this top-level description clearer than the existing documentation of the system?

The resources to be devoted to this project should be less than one man-month of senior analyst time, including the reporting and evaluation.

4.2 CISS THREADING

The recurring critical question to be addressed in this experiment is: Can the requirements for a BMD problem be described in terms of a set of data flow paths (or threads) through the software? And are the number of such threads manageable? Reference 7 presents the Computer-Independent Software Specification for the Terminal Defense Program and is currently the defining document for the process design being performed by Texas Instruments; this is a known, realistic BMD example.

Since the CISS exists, but the SRPS is suspected not to contain sufficient information to derive the CISS, there will be no attempt to trace the SRPS requirements into the requirements document. For the same reasons, this experiment will not address the derivation of the software performance requirements; its scope will be restricted to the description of the requirements in the existing CISS.

The amount of resources devoted to this is approximately two man-months, excluding evaluation of the results. A portion of the evaluation will include interaction with TI to obtain their evaluation of the resulting product.

4.3 TRACK LOOP

This experiment will provide the example for inclusion into the SRPS and CISS definition documentation and into the methodology description as to how the CISS performance requirements are derived from the SRPS information. Since the existing TDP SRPS may not contain a sufficient set of information, this example will be constructed in order to demonstrate the traceability of SRPS to CISS, but may not start from the existing SRPS for TDP.

The basic documentation for the experiment is a subset of [7] which deals with the tracking of objects. If necessary, the CISS Simulator will be modified in order to determine software performance requirements. When the complete example is available in rough form, the results will be shown to GRC for comment on their assessment as to the depth of the SRPS implied, and to TI for comment on the adequacy of the resulting CISS. The experiment should take between two and four man-months for its performance.

4.4 CISS SIMULATOR PERFORMANCE ANALYSIS

The purpose of this collection of analyses is to determine the performance requirements of specific types of BMD processing using the CISS [7] and the results of the CISS Threading Experiment as a baseline. Questions to be addressed include: What is the proper form of the performance requirements relating to the Resource Management functions? If simulation is necessary in order to determine the effects of computational inaccuracies, the CISS Simulator is to be used if possible. The totality of all such analyses should be on the order of three man-months.

4.5 ADVANCED BMD ALGORITHM EVALUATION

TRW will, as directed by BMDATC, perform preliminary evaluations of selected advanced BMD algorithms. Where appropriate, this may be extended to the preliminary analysis of the requirements which might be the result of adding such an algorithm to the software system described in the CISS. This will provide a good example of how a system described by the methodology might be modified according to the methodology.

The algorithms to be addressed, as well as the constraints on the effort to be expended, are the subject of future direction by BMDATC.

4.6 AUGMENTED STOPLIGHT PROBLEM

The purpose of this experiment is to obtain an example to be used for the communication of the methodology and to investigate the feasibility of different techniques for automated simulator construction. The example will be an augmentation to the previously documented Stoplight problem and will cover the range of SRPS to CISS, including the development of a simulator.

The resources to be expended should be on the order of six man-months, including analysis.

4.7 SMALL EXPERIMENTS

This collection of analyses is included for several purposes: To allow for future definition of experiments not presently identified; to allow for the case studies to be performed as part of the analysis of the applicability of the methodology to non-BMD problems; to provide a place for the experiments whose purpose will be to construct test cases for the tools; to identify "measures of success" for the formal evaluation experiment; etc.

For example, if someone performs an analysis of a previously documented non-BMD system in order to determine the applicability of the methodology to the specification of software requirements on the system, its results should be included as an experiment.

The resources and schedules are to be specified.

4.8 FORMAL EVALUATION EXPERIMENT

This refers to a formal demonstration of the methodology to be performed in 1976. A SRPS will be produced by GRC and the results of the analysis will be provided to a Process Designer. A Test and Evaluation Procedure will be written (currently due April 1975) before the experiment is started.

5.0 REFERENCES

1. "ABMDA Research and Development Software Standards," SDC, 8 September 1972.

2. "ASSD Software Performance Requirements - Software Requirements Engineering Methodology," TRW Report No. 22944-6921-011, August 15, 1974 (CDRL G009, DAHC60-71-C-0049).

3. "ASSD Software Capability Description - Software Requirements Engineering Methodology," to be published 1 October 1974 (CDRL G00B, DAHC60-71-C-0049).

4. F. T. Baker, "IBM's Chief Programmer Team Experiment," IBM SYSTEMS Journal, Vol. II, No. 1, 1972.

5. F. T. Baker and H. D. Mills, "Chief Programmer Teams," Datamation, December 1973.

6. F. Mertes, et al., Final Report, "Automation Applications in an Advanced Air Traffic Management System," TRW Systems Report No. 22265-W008-RU-00, December 1973 (5 volumes).

7. "Computer-Independent Software Specification for the TDP Baseline Construct," TRW Systems Report No. 22944-9771-RE-00, May 1974 (CDRL G00E, DAHC60-71-C-0049).

22944-6921-016

SOFTWARE PERFORMANCE REQUIREMENTS-
SOFTWARE REQUIREMENTS LANGUAGE

CDRL G00G

OCTOBER 1, 1974

Sponsored By
BALLISTIC MISSILE DEFENSE
ADVANCED TECHNOLOGY CENTER
DAHC60-71-C-0049

TRW
SYSTEMS GROUP
Huntsville, Alabama

SOFTWARE PERFORMANCE REQUIREMENTS-SOFTWARE REQUIREMENTS LANGUAGE

CDRL GOOG

OCTOBER 1, 1974

Sponsored By
Ballistic Missile Defense
Advanced Technology Center
DAHC60-71-C-0049

TRW
SYSTEMS GROUP
Huntsville, Alabama

SOFTWARE PERFORMANCE REQUIREMENTS-SOFTWARE REQUIREMENTS LANGUAGE

CDRL G00G

OCTOBER 1, 1974

Principal Authors

T. Bell
E. Benoit
D. Bixler
M. Dyer
J. Richardson
W. Smith

Approved By:

T. W. Kampe

T. W. Kampe, Manager
Software Requirements
Engineering Program

James E. Long

James E. Long, Manager
Huntsville Army Support Facility

Sponsored By
Ballistic Missile Defense
Advanced Technology Center
DAHC60-71-C-0049

TRW
SYSTEMS GROUP
Huntsville, Alabama

TABLE OF CONTENTS

LIST OF FIGURES

1.0 INTRODUCTION

The Ballistic Missile Defense Advanced Technology Center (BMDATC) has established the Software Development and Evaluation Program in order to develop a complete and unified engineering approach to software development and testing ranging from synthesis of system specifications through completion and testing of the process design. A key element in this program is the Software Requirements Engineering Program (SREP)--a research program concerned with the development of a systematic approach to the development of complete and validated software requirements specification [1]. Key program objectives are to ensure a well-defined technique for the decomposition of system requirements into software requirements, provide development visibility, develop the requirements independent of machine implementation, allow for rapid response to system requirements changes, and provide testable and readily validated software requirements.

In addition to the methodology for requirements development and specification, an integrated software system, Requirements Engineering and Validation System (REVS), will be developed to support use of the approach developed. As a result of current and previous work, a basic structure for defining and communicating requirements has been identified which lends itself to automation. Specifically, the requirements can be stated in terms of sequences (called threads) of processing steps (called alphas) each of which represents the necessary processing required to achieve a desired data processor response to a particular stimulus under a given set of conditions. Thus, the software system will automate certain procedures in the methodology with the actual software requirements forming the primary data base for the procedures and aid humans in performing other procedures. The requirements, which are derived from system requirements stated in a System Requirements and Performance Specification (SRPS), are decomposed, analyzed, and validated as Process Performance Requirements (PPR)[1].

---

1. Though specific contents of the PPR is currently being defined, the level of detail would be analogous to the Computer-Independent Software Specification (CISS).

In conjunction with and parallel to the methodology and REVS capability definition, a Requirements Statement Language (RSL) will be defined consisting of syntax and semantics necessary to define software requirements (e.g., PPR content) and to control REVS. Even though a profileration of languages currently exist, no one language possesses the syntax and semantics necessary in requirements engineering for large scale real-time software developments as characterized by Ballistic Missile Defense (BMD) systems[2]. What is needed is a computer processable language that

- forces structure and discipline in the requirements analysis process
- minimizes information transfer without over specification of design
- reduces ambiguity and contradiction in the specification of design
- minimizes the amount of manual work required to produce all of the specification related products
- controls REVS.

The language must allow requirements to be stated in terms used by the requirements analyst rather than terms used to program computers (i.e., a user oriented language rather than programmer oriented language like FORTRAN). It must be applications oriented only to the extent necessary to handle BMD systems, and non-procedural to the greatest possible extent.

The purpose of this document is to define the fundamental concepts, general requirements, and desired characteristics of that language for expressing software requirements and controlling an integrated requirements engineering and validation system. The information content of software requirements and the capabilities of the REVS software are not yet finalized, but general requirements for language syntax and semantics can still be specified. For example, detailed requirements are presented which allow for alpha definition, thread definition, global data base definition and performance requirements specification. As the definition of the information content of software requirements evolves with the methodology, these requirements

2. An analysis of capabilities offered by existing languages is presented in Current Software Technology [2].

will necessarily be modified. The detailed requirements syntax and semantics for REVS control will be documented later in the research program after a preliminary design of that software has been performed. Until a preliminary design is completed, only concepts for REVS processors exist. For example, in supporting thread creation and management, REVS processors could perform text editing, library management, configuration control and requirements tracing. Supporting requirements analysis and validation, other processors could analyze and synthesize a global data base, synthesize thread trees, and construct and execute simulators and/or emulators. From these concepts, however, general language requirements were derived which insure a unified RSL that addresses both requirements definition capability and REVS control.

Section 2 of this report contains a list of applicable documents. Sections 3, 4, 5, and 6 address the approach to language development, general requirements on the language, specific requirements on the semantics for requirements definition and techniques for language implementation, respectively. Section 7 contains a glossary of nomenclature used within this report.

The RSL and REVS software will address systematically, and provide automation for, a critical part of the software development process. When developed and operating synergistically with the Texas Instruments (TI) Process Design System (PDS), BMDATC will have provided a software design system which addresses the spectrum of software development activities--from requirements definition to process design.

2.0 APPLICABLE DOCUMENTS

1. "Software Performance Requirements - Software Requirements Engineering Methodology," TRW Report 22944-6921-011, Contract DAHC60-71-C-0049, CDRL G009, 15 August 1974.

2. "Current Software Requirements Engineering Technology," TRW Report #22944-6921-010, Contract DAHC60-71-C-0049, CDRL G00A, to be released.

3. "Software Capability Description - Software Requirements Engineering Methodology," Contract DAHC60-71-C-0049, CDRL G00B, to be published.

4. "Errors-Detection and Correction," O. N. Freeman, et al, Compiler Techniques, B. W. Pollack (Ed.), Averbach Publishers, 1972, pp. 270-340.

5. "Concern for Correctness as A Guiding Principle for Program Composition," E. W. Dijkstra, The Fourth Generation, Infotech, 1971, pp. 357-367.

6. "The Complexity Programs," H. Mills, Program Test Methods, W. C. Hetzel (Ed.), Prentice-Hall, 1973, pp. 225-238.

7. "An Automated Tool for Extended Static Error Detection," R. W. Smith, TRW Memorandum 74-2532.7-5, February, 1974.

8. "Design and Construction of An Automated Software Evaluation System," C. V. Ramamoorthy, R. E. Meekey Jr. and J. Turner, IEEE Symposium on Computer Software Reliability, 1973, pp. 28-37.

9. "Automatic Generation of Self-Metric Software," L. G. Stucki, IEEE Symposium on Software Reliability, 1973, pp. 94-100.

10. "Evaluating The Effectiveness of Software Verification - Practical Experience with An Automated Tool," J. R. Brown and R. H. Hoffman, Proceeding of FJCC, 1972, pp. 181-190.

11. "Optimal Software Test Planning through Automated Network Analysis," K. W. Krause, R. W. Smith and M. A. Goodwin, IEEE Symposium on Computer Software Reliability, 1973, pp. 18-22.

12. "TRW/TM* Programmer's Guide," B. Press and P. Courey, M1025-00, TRW Systems Group.

13. "The STAGE2 Macro Processor," W. M. Waite, Tech. Report 69-3, Graduate School Computing Center, University of Colorado, Boulder, 1969.

14. "ML/I User's Manual," P. J. Brown, University of Kent at Canterburg, June, 1967.

3.0 LANGUAGE DEFINITION APPROACH

The conventional approach in language design has been to implement a proposed design, wait for several years to determine the most critical problems, and then reimplement a revised design. Incremental changes to the language are not often made because the costs of implementation are simply too large. This conventional approach is unacceptable in the context of the SREP effort because we know too little about the relative power of different parts of the new methodology to be sure of being correct with our first design, and because we have neither the resources nor the time to "hand code" the translators, compilers, service routines, etc. that would be required to make several formal attempts at languages.

The approach to be pursued in defining the RSL is to use a number of advanced techniques to implement translators as described in Section 6. By using powerful software tools in the design and implementation of the RSL, we will be able to break through the existing problems of cost and inflexibility. Our approach is to perform simple, manual translation jobs to ensure that our syntax and semantics are unambiguous and complete to the extent possible, and then to implement a tentative design cheaply. Using meta-compilers and/or macro processors, we will then be able to revise the language rapidly and cheaply. We will use a language orientation that facilitates extendability so that our final output will not restrict the users, and our intermediate results can be modified through extension as the RSL is employed in experiments. In these ways the RSL will be incrementally refined and not suffer from the definitional difficulties so often encountered with new languages.

Frequently, language implementation begins too early; the implementer begins with a personal concept of a specific design and preceeds to implement it. The result is that an unstated or nebulous set of requirements are met, but important, unstated requirements are ignored. Instead, the implementer should begin with a set of well-stated requirements that can be revised as he learns the implications of his objectives. Ideally, these objectives and requirements should be stated initially only after some experimentation with tentative ideas about the characteristics of the ultimate language.

The rate of change in requirements can then be reduced and poorly conceived ideas discarded before designers create a "mental set" that limits their innovative capabilities.

The approach utilized for language definition will necessarily be an iterative one where requirements are derived from evolving Software Requirements Engineering Methodology (SREM) concepts. Basically, the RSL objectives are derived from two sources within SREM: the PPR content definition and the REVS software capability definition. As new concepts and capabilities are defined and developed, the syntax and semantics will be augmented. To ascertain the language requirements, language definition activities must be accomplished in parallel with the SREM concept definition.

The most critical single requirement on the language syntax and semantics is that of representing the PPR content. A preliminary set of RSL semantics requirements have been defined, Section 5, based on current concepts regarding PPR content. As indicated previously, a close interaction with methodology development will be required to insure completeness and consistency. Particularly in this area of language definition many trade-offs will be required to resolve potentially conflicting requirements (e.g., readability and naturalness vs ease of use and reduction of ambiguity). The issues of readability and naturalness are of particular interest to the requirements analyst as the language is to be the communications medium to the process designer as well as the REVS data base input medium.

In parallel with the REVS software capabilities definition, language syntax and semantics will be defined which invoke the processors and provide any other inputs required by that processor. Although certain processors may require specialized syntaxes, all will be made to conform to a single overall language philosophy depicted in Figure 3-1 and perform certain functions (e.g., error recovery) similarly. An example of a syntax for a single unified language system which allows users to interface with REVS processors is shown in Figure 3-2. The syntax shown, although not to be construed as a candidate under consideration, has desirable attributes some of which will be specified as general syntax requirements in Section 4. The intent here is to show a single input stream which controls REVS processors in a non-procedural manner. Note the optional use of articles and English clause forms which

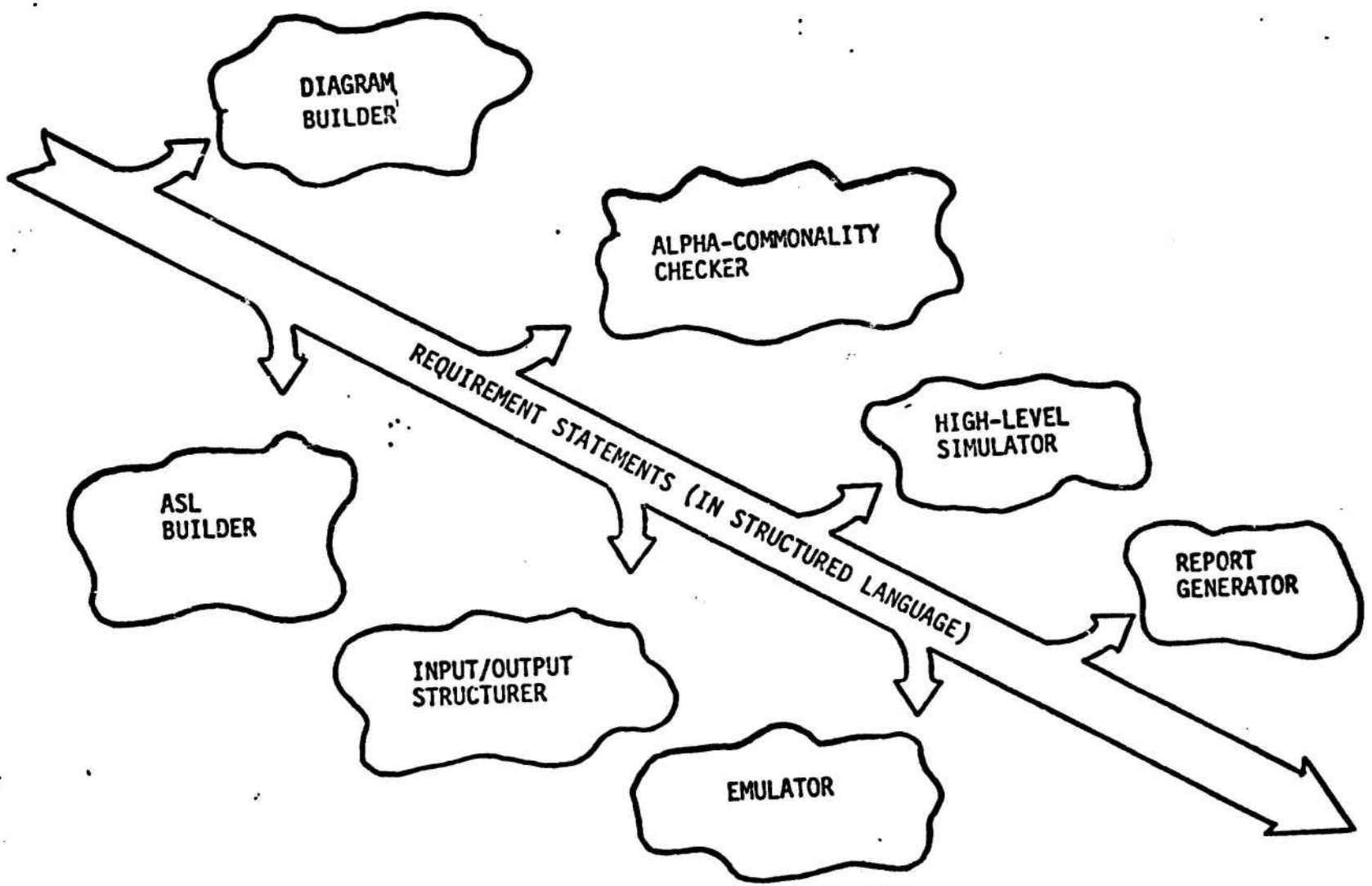


Figure 3-1 Integrated Language System

DEFINE AN ALPHA name or DEFINE ALPHA name
(alpha definition syntax)

STORE ALPHA name IN LIBRARY name WITH READ ONLY PERMISSION

FETCH ALPHA name FROM LIBRARY name

EDIT THE ALPHA
(alpha editor syntax)

REPLACE ALPHA name WITH ALPHA name IN LIBRARY name
or
REPLACE ALPHA name, name, LIBRARY name
or
REPLACE ALPHA name IN LIBRARY name WITH ALPHA name

DEFINE THREAD name
(thread definition syntax)

STORE THREAD name IN LIBRARY name

GENERATE N2 DIAGRAM FOR THREAD name FROM LIBRARY name

SYNTHESIZE ASL
(ASL synthesis syntax)

STORE ASL name IN LIBRARY name WITH NO PROTECTION

CONSTRUCT SIMULATOR FROM ASL name STORED IN LIBRARY name WITH LIBRARIES name, name,..., AND name
(additional simulator construction syntax)

STORE SIMULATOR AS name IN LIBRARY name

EXECUTE SIM name, LIBRARY name, OUTPUT FILE name (short form)
(simulator input syntax)

ANALYZE FILE name
(post processor input syntax)

Figure 3-2 Integrated Requirements Language Structure

enhance readability and at the same time provide for ease of use. Also note that positional notation is omitted entirely and that key words are used minimally. In an effort to reduce or eliminate redundancies in software development, examination of existing systems initiated earlier in the SREP effort is scheduled to continue during the preliminary REVS design phase. In addition to possibly providing processors which can be interfaced to the user through the RSL, language concepts will be studied with the intent of incorporating those which increase the acceptability of the product. For example, PDS integrated language concepts, TRW-TSS interactive features, JSL, JCL or SCOPE processor control techniques, and COBOL language partitioning could be adapted to the RSL.

Finally, investigations will determine the effect on language syntax and semantics of operating in a batch vs interative environment with the intent of providing both capabilities within the RSL. In conjunction with this activity, ARC resources will be investigated to identify constraints which may limit language implementation. Such issues as character sets, mass storage resources which could limit library management schemes, and BMDATC standards on software deliverables are examples of constraints which need to be addressed.

The combination of beginning with specific objectives and implementing with powerful tools will enable us to include steps in the approach that are usually impossible. The language can be partitioned into conceptual pieces, the interfaces between pieces defined, new concepts evaluated, and the implications of both interactive and batch processing analyzed.

4.0 GENERAL REQUIREMENTS ON THE LANGUAGE

A set of general requirements has been derived which will insure definition of a single unified language system syntax for software requirements definition and REVS control. The syntax requirements provide for an understandable, requirements analyst oriented syntax (i.e., provides a communication medium between the requirements specifier and the process designer) and is applications oriented only to the extent necessary to support BMD software specifications[1]. The general processing requirements for the language imply certain capabilities for the REVS software and, in fact, represent objectives or goals for REVS software design. These requirements address those REVS capabilities which would not necessarily be determined from analyzing the processing steps in SREM and could possibly apply to several steps by providing common requirements (e.g., error mode termination). Some of these latter requirements are vague as they are dependent on the ARC facilities to be utilized for development and BMDATC direction. These requirements will be modified as required during preliminary design and are included here for completeness.

4.1 GENERAL SYNTAX REQUIREMENTS

The following paragraphs describe syntax attributes which must be considered when defining the RSL. As some attributes present somewhat conflicting requirements, trade-offs will be required during the language definition activity.

Natural. The RSL must be natural to the BMD analyst by providing convenient ways to indicate such things as vector and matrix notation, subscripts and superscripts, and other mathematical operators in a traditional manner.

Understandable. The RSL must be a readable and understandable language. The definition of the language syntax, semantics, and structure should facilitate the understanding of what is being specified by the following groups:

---

1. Reference 3 presents a list of assumptions regarding the nature and characteristics of the systems to be addressed with SREM which also pertains to and constrains the RSL definition.

- The customer who requested a PPR.
- The requirements analyst(s) using RSL.
- Other requirements analysts.
- The requirements analyst's managers.
- Simulator/Emulator developers.
- The process designers who will use the PPR.

Understandability [5,6] will be enhanced by:

- The nomenclature specification should be flexible. For example, a restriction should not be placed on a RSL user that forces all identifiers to less than seven characters.
- The delimiters used for punctuation of the language should be English-like. For example, a period should be used to end an RSL statement and not a dollar sign.
- The RSL should not allow the same thing to be done in many different ways. A tradeoff exists here between ease of use and flexibility.
- The language should be structured in such a way that it is readable from left to right and top to bottom.

Also, if the REVS software should ultimately support creation and use of libraries of common BMD functions, the syntax for their use must be familiar and meaningful to the analyst.

Ease of Use. The RSL syntax must have characteristics which promote ease of use and make it difficult to make mistakes. Some considerations are listed below:

- The RSL should allow for free-form user inputs, e.g., the user should not be required to input certain items beginning in particular columns.
- The RSL should contain a minimum number of keywords, but when a tradeoff exists between keyword and positional parameter techniques, the keyword technique should be adopted.
- The use of abbreviated inputs should be allowed.

To satisfy the objective of a unified RSL and maintain ease of use, one input stream of RSL statements will contain all the information required from the user to specify his requirements and to invoke any or all support software. This implies that the syntax must be defined with distinguishing characters, punctuation, and/or structure which facilitates user recognition of definition, analysis, and validation processing steps. Also the RSL should be formatted to facilitate easy identification and alteration of sequences in which ordering is important (i.e., sequence of alpha's forming a thread).

4.2 GENERAL LANGUAGE PROCESSING REQUIREMENTS

The general requirements presented in the following paragraphs necessarily impact the REVS software capabilities description and design to be performed in latter phases of the research program [1]. However, as these requirements emanate from language translation and processing, they could be overlooked when designing REVS software to support aspects of SREM procedures. The requirements apply to the spectrum of REVS software enhancing the probability that common procedures and techniques will be utilized.

Syntax Diagnostic Requirements. The RSL processors should have extensive diagnostic capabilities [4]. In generating diagnostics, the syntax processor should be designed to provide the following characteristics.

- All diagnostic messages should be meaningful to the user in terms of source code.
- Sufficient detail must be given in each diagnostic such that a reference to some manual by the user is not necessary.
- The translator should detect simple syntax errors such as misspelling and automatically correct them.
- Sufficient information should be provided by the processor to indicate source of errors.

Static Diagnostic Requirements. The RSL processors should perform extensive static error analysis [7, 8] in an attempt to identify logic errors. Some potentially detectable logic errors are:

- The use of a parameter as an input without any possible way of assigning a value to the parameter.
- The assignment of a value to a parameter without any possible future use of the parameter.
- The definition of processing steps that are not allocated to processing sequences.
- The specification of conflicting initial conditions for exercising a processing step.
- The occurrence of an infinite loop because of conflicting conditions for termination.

The RSL should produce a variety of cross references to enhance the detection of errors in a static mode. Some useful cross references are N^2 charts, alpha dependency charts, and detailed global data cross references. Other possible static processors include a test data design aid, a test case library manager, and a test results comparator.

Dynamic Diagnostic Requirements. These RSL processor requirements are directed toward facilitating validation of software requirements through simulation and emulation. The RSL processors should provide a variety of dynamic test and debug aids [9, 10, 11] to be applied during execution of the simulators (emulators). One aid to be considered is a program execution monitor which could (1) trace program executions, (2) record items executed by different tests, (3) detect and isolate errors (e.g., data manager errors, enablement criterion violation, etc.) and (4) provide a snapshot dump in readable formats as selected by the user.

Error Handling Requirements. Two requirements pertaining to diagnostic processing are that detected errors should not terminate the syntax scan nor necessarily terminate processing. Processors should, where feasible, continue syntax scan diagnosing errors. Also, when processing in a batch environment, control transfer from processor to processor should be conditional on the nature of the errors detected in prior phases. This latter requirement is applicable to the interactive mode, but must be extended to require all processing terminate at the user interface for further instructions.

Interactive Requirements. Special requirements must be specified to insure a common user interface. Processors interfacing with the user interactively through RSL should provide for:

- User interrupts calling for immediate termination of processing
- Processor restart capability (simulators/emulators, etc.)
- Acknowledgment of message receipt
- Termination indicator (i.e., request for next input)
- Prompting mode of input where at each opportunity for input, the user is informed of his options
- Abbreviated form of inputs where strings of RSL statements are accepted and processed without prompting

- Interactive report formatting

These requirements also impact syntax definition.

Report Generation. The RSL processors must incorporate flexibility to organize and generate requirements documentation. Processors providing these capabilities should provide:

- Automatic consistent formats (i.e., column oriented with indentation)
- Automatic consistent spacing within and between RSL statements, headers, and titles
- Replacement of abbreviated terms with long forms
- Insertion of titles and headers

The RSL processors providing simulator data analysis and performance requirements generation should provide additional capabilities to produce

- tabular, or graphic outputs
- alternate data formats
- alternate data collection
- data comparisons from multiple runs

5.0 SEMANTICS OBJECTIVES FOR REQUIREMENTS DEFINITION

The language objectives are to provide the syntax and semantics necessary for stating requirements for the phase of software development addressed by SREM. Current SREM concepts, though evolving, have identified procedures for determining requirements [1] and the structure and information of those requirements to be communicated to the process designer. Current studies have determined that the desirable attributes of good requirements [1] can be obtained by specifying the processing in terms of structured processing threads.

A processing thread is a means of stating a subset of the total requirements in the classical input-transformation-output format. All such threads for a given system comprise the complete set of system (or software) requirements. A thread is a single linear processing path through the system between system interfaces or nodes. (The most frequently used nodes are the input and output ports; however, internal nodes can also be used.) A thread may be represented pictorially as shown below:

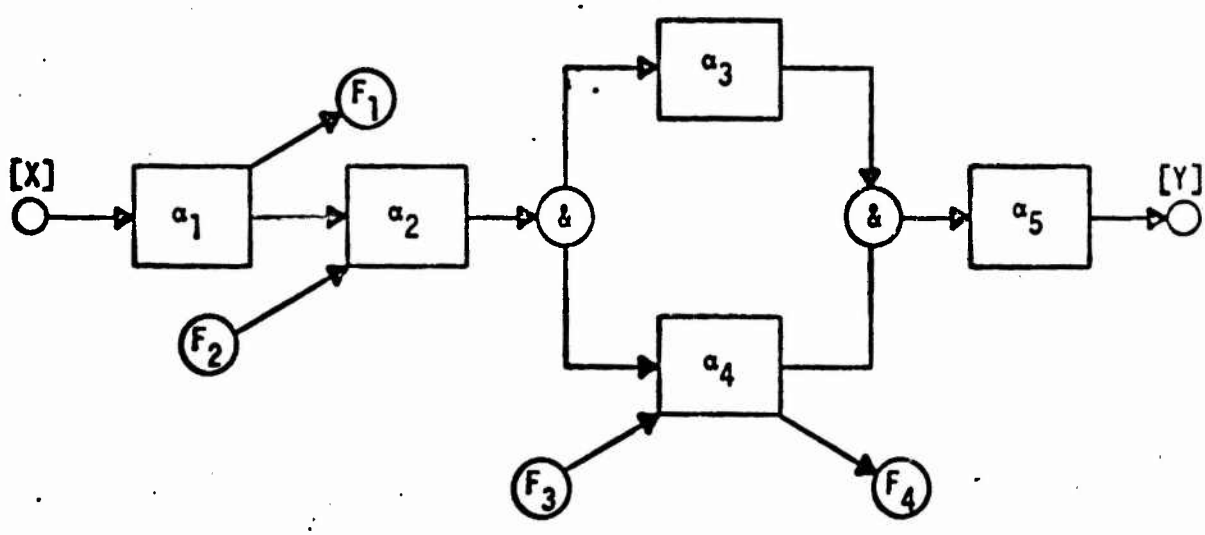


Figure 5-1 Threads Concept

where:

[x] is the input data set which consists of:

- A stimulus - a physical occurrence which leads to the need for processing (e.g., the arrival of a message, a previously determined time, an interrupt, etc.).
- Any input data associated with the stimulus including admissible range of values and expected error distribution.

α_i is the i^{th} processing step which is required and constitutes an element of software requirements. An α is characterized by input data, output data, and a precise statement of the transformation function. This transformation function may be a logical (Boolean) operation, mathematical operation, or a combination of both. It may be stated functionally or analytically, depending upon the level of detail desired.

[y] is the required output response--typically an event or stimulus to an external device with an associated data set.

F_j is the j^{th} file in the global data base (GDB).

GDB is the set of all data which is required to be saved within the system. Data is required to be saved for one of two reasons: (1) it is required at a future point in time for a subsequent execution of the software representing the thread, or (2) it is required as input to a processing step in some other thread.

The processing steps must be performed in the order indicated in the drawing except where the symbol & or + is used. The symbols used in the example above mean that the α_3 and α_4 processing must be done after the α_2 processing is completed and both must be completed before the α_5 processing. No restrictions are imposed on the order of processing of α_3 and α_4 relative to each other. If the symbol just before α_5 were a + , this would mean that α_5 processing could proceed when either α_3 or α_4 is completed.

Additional attributes of the thread which must be included are the following:

- Arrival rate of input stimuli. For a system subjected to many similar stimuli from different sources (e.g., tracker/fuse correlation algorithm with inputs from two or more sensors), this is

  λ = arrival rate of the stimulus from one source

  N = number of sources generating stimuli

  then $N\lambda$ = total arrival rate of stimuli to the thread

- Time Response. This is the allowed time between the arrival of the stimulus and the availability of the output response.

- Accuracy. The allowed error in the output response data must be defined. This accuracy requirement must be carefully and precisely defined if it is to be testable.

- Conditions. The conditions under which the thread is to be executed must be defined.

In addition to those structured aspects of requirements, some items in a specification may not be clearly a portion of any structural part of a candidate system. These non-structured requirements may be ones that are yet to be allocated, may be ones that cannot be allocated until software implementation, or may have been drawn together for clarity. These non-structural aspects constitute another area which is dealt with separately.

Determining requirements that constitute the PPR is a complex design process, and the language to support the process defined in the methodology will be heavily oriented toward supporting design. However, the environment that overlays this design process is the statement of requirements, and this creates the need for certain management-oriented characteristics such as traceability of requirements from one level to another. Some of these needs have already been integrated into the areas noted above, but the implications of others remain undefined at this time and are given in a separate final section.

5.1 ELEMENTAL ASPECTS OF REQUIREMENTS (α's)

The elements on the structure (however represented) constitute requirements that must be expressed. The elements must be easily and uniquely identifiable in order for designers on different parts of an effort to work effectively. They must also have textual names so that their meaning is clear without reading a piece of text several paragraphs long. In addition to these objectives to ease interpretation by humans, an element's detailed meaning must be understandable by the computer so that this meaning can be used in functional simulations and emulations.

Empirical evidence gained during experimentation has indicated that some elements simply cannot logically be placed on threads. These elements have weak interactions with a number of threads but strong interactions with none. Their cumulative effects on the system are so large that they cannot be ignored, so they must be represented. Similarly, designers sometimes decide to specify queues in a system so that interface requirements can be met by choosing from the queues in adroit manners. These queues and their servicing rules must be expressed along with the global elements and the elements that fall nicely on threads. These characteristics combine to yield the following objectives.

1. Provide a unique numbering scheme for elements.
2. Provide textual areas for input of design notes.
3. Show analytical transformations--both functionally and in detail. (This will probably require procedural statements so that emulation and functional simulations can be performed.)
4. Allow for English names which can be searched for replacement by standard alphas.
5. Indicate queue servicing functions when these are complex and have functional requirements.
6. Provide a means for indicating that an alpha is a portion of one or more higher level alphas.
7. Show elements that are not on chains (global ones).

5.2 STRUCTURAL ASPECTS OF REQUIREMENTS

The structure of a system is usually represented by a hierarchical breakdown chart to aid in assigning personnel to logical pieces of it. An alternative is to adopt the basic approach evident in much of the methodology--to structure the system by identifying the flows through it. These flows are what we have termed threads because they cause the initiation of processing or constitute a transaction that flows through the system.

Early experimentation with actual cases has indicated that certain realistic problems exist with structural representation. For example, we have found that the number of threads can rapidly become so large that the designer cannot easily maintain context; some technique (e.g., thread trees) must be included to allow for indicating that two chains are identical with the exception that different processes may be executed at a single point in the flow. The combination of these specific, realistic problems and the general need for representing structural aspects of requirements in several ways leads to the following initial set of objectives:

1. Show threads (sequences of alphas).
   a. Caused by any means
   b. Queued and triggered somewhere else--this may be an important case since it is the most common means to meet maximum load requirements; the queue service discipline may be specified for this. An example is the radar scheduler.
2. Indicate parallelism (concurrency, don't care sequences).
3. Indicate ORs (which is a technique for realistically dealing with mulitple, similar threads).
4. Provide a means for optionally indicating the hierarchical, conventional structure of alphas.
5. Provide an easy vehicle to the language from an intermediate notation for expressing interactions. (This may not be realizable and thus requires a processor; it probably is not so important an objective as other ones.)

5.3 INPUT/OUTPUT ASPECTS OF REQUIREMENTS

Inputs are, of course, important to elements and must be stated if a software implementer is to correctly design a data base and implement an alpha as intended. Inputs are also important in the statement of requirements because they indicate the conditions under which a thread will be activated. Both of these basic needs result in objectives for representation in the language as indicated below.

1. Indicate each required input for computation to an alpha.
2. Indicate all decisions input to each alpha.
3. Indicate all meaningful outputs from an alpha (including updates to the data base and data that may be passed to other functions as parameters if not part of the data base).
4. Indicate if a particular instance of the data is required, or if the latest value will be used without respect to its time of update.
5. Show the approximate number of instances and the magnitude range for each data element in the data base.
6. Indicate all inputs (coming from a port or due to a specific update) that are required before a function can be performed.
7. Indicate the triggering input for each thread and its source of data.
8. Indicate the data (and source) for each choice between subthreads at an OR.
9. Show the frequency of occurrence of input triggers to each thread.

5.4 TIMING ASPECTS OF REQUIREMENTS

Timing is not meaningful for most specific elements in the functional (as opposed to real-time software) domain. However, timing is of great importance for many groupings of elements and must therefore be specified in the language. All the objectives involving timing requirements, therefore, address timing along threads and only indirectly involve elements. Ideally, all timing requirements would specify periods from external input ports of threads to external output ports. However, restricting timing requirements to address these cases only is inadequate since, for a number of actual cases encountered during initial experimentation, the most appropriate time constraint begins or ends at a data base update. Such update points are called internal ports.

Some time constraints are absolute; they must be met absolutely every time or the system's integrity is lost. Quite likely this assumption is adequate for the early phases of developing the PPR, but in the later phases this assumption can be relaxed for many cases since probabilistic constraints are more appropriate. The objectives below reflect this situation.

1. Indicate the initiating and terminating conditions for a timing requirement. This might include:
   a. A trigger for a thread coming over an external port.
   b. A set of conditions including an internal port.
   c. Completion of an alpha's processing.
2. Accept a set of times (including a set containing only one item) and an optional set of conditions (e.g., this processing must be completed within 10 msec if the track rate is 10 per second and within 5 msec if the track rate is 40 per second).
3. Accept a set of probabilities--indicating the required portion of instances that the specific time limit must be met.
4. Provide a textual area for indicating the techniques for testing the real-time code to ensure that it meets the requirements.

5.5 ANALYTICAL ASPECTS OF REQUIREMENTS

Analytical requirements may be appropriate between either internal or external ports, but may also apply only to a portion of such a thread. This is true because, as successive functions are performed, outputs become dependent on increasing numbers of inputs, and relationships become more difficult (perhaps infeasible) to state. Complicating this problem is the possibility of employing computational techniques that meet the requirement only probabilistically. The longer the chain of such computations, the more difficult will be the selection of appropriate limits.

The system-wide effects of some analytical computations may be unclear; designers may have found through experimentation that a particular functional form of output causes the system to perform well while others do not. Such cases must be noted explicitly so that software implementers know which functions are crucial and which are "for information only". Given this information, appropriate controls can be instituted.

These and a variety of other needs lead to the following objectives:

1. Indicate an arbitrary set of sequential processing steps (including sets with only one member) for which the analytical requirement will be stated.

2. Indicate the constraints (limits) on the variable set (in functional or tabular form).

3. Indicate the portion of time that values may fall outside the limits.

4. Provide a text area for indicating the techniques for testing the real-time code to ensure that it meets the requirements.

5. Indicate the type of requirement (natural law, case of "truth" being known only approximately, heuristic techniques, or case of alternative interpretation).

6. Indicate the level of control for the set. (Tentatively, three levels of control might be employed -- 1) specific requirements must be met precisely with no alternative interpretations allowable, 2) alternative forms allowable but they must be approved by PPR developers, and 3) details may be changed.)

7. Indicate the set of output variables to which the requirements apply.

5.6 NON-STRUCTURAL ASPECTS OF REQUIREMENTS

The human mind does not always work logically, and ideas may pop into a designer's consciousness without respect to the design structure we define. The language must enable him to state and deal with requirements of this sort. For example, a designer might realize that all inputs coming through a port have the possibility of being erroneous in a certain, identifiable way. This class of errors is important, and each thread should include an element at its front to correct the errors. The initial conceptualization of the requirements is not associated with each thread, and communicating the requirement to an individual concerned with data correctness would be difficult if he had to see every thread. Instead, stating the requirement separately from the structure (the definition of threads) would yield real utility.

Another general class of requirements can be stated most easily non-structurally: Those requirements that pass from the system specification directly to the PPR for implementation in software. An example might be a system requirement for security which could be fulfilled by implementing the software with a small, hardened nucleus. This requirement would need to be passed to the software implementers, but it would not be reflected in the areas of requirement aspects noted above.

System requirements analysts may desire (or contracts officers may demand) that some requirements be stated non-structurally for clarity in large systems, particularly in the early stages when ambiguity may be high. For all these reasons, the following objectives exist:

1. Allow a unique identifier to be given for each requirement.
2. Indicate a priority for meeting the requirement (so that tradeoff studies can be performed when inconsistencies or cost implications become known).
3. Provide a text area describing the requirement.
4. Allow English naming of the requirement.
5. Provide a technique for indicating the threads (or elements or functional items) to which the requirement is allocated.
6. Provide a text area for describing the influence of the requirement (if it is not allocated to anything specifically).

5.7 MANAGERIAL ASPECTS OF REQUIREMENTS

Many of the traceability needs of system design have been previously stated in other areas. For example, the section regarding non-structural aspects of requirements provides for traceability of those requirements. However, the allocation of a requirement from a higher level thread to a lower one has not been explicitly addressed because we are not yet sure whether this should be done differently for each aspect of requirements or for the threads themselves as entities. Some technique is needed, but we do not yet know how to allocate this objective among the different areas.

Several other attributes of requirements (such as their criticalness to system integrity and the degree to which they have been validated) need to be indicated. These general attributes of requirements may prove to be most easily dealt with from a global viewpoint for defining the language. The specific objectives of this kind are as follows:

1. Provide a technique for indicating the criticalness of each requirement to system integrity.
2. Provide a technique to indicate whether a requirement is an allocated or derived requirement.
3. Show the traceability of requirements from the highest level to lower ones.
4. Indicate the degree of technical validation of each requirement.
5. Indicate the degree of formal approval of each requirement.

5.0 FUTURE CONSIDERATIONS

The objectives stated previously may be revised as further experimentation indicates that they are not necessary or that they require modification. They will be subject to further revision when the realities of implementation indicate the relative costs of each. In addition, trying to meet too many objectives in a single language can make it unwieldy to use; the objectives of usefulness is an overriding one and may cause us to temper many of the objectives previously stated.

We hope that in many cases the same language design elements will fulfill several of the objectives. Clearly a new verb for each of the requirements would create a huge set of keywords and make implementation and use unreasonably difficult. Judicious employment of good language design and deletion of less important objectives will be necessary to arrive at a good solution.

6.0 IMPLEMENTATION TECHNIQUES

The approach to be pursued in developing the RSL as outlined in Section 3 is to use a number of advanced techniques to implement translators. By using powerful software tools in the implementation of the RSL, it will be possible to hypothesize a candidate language, evaluate it with respect to the requirements, and modify it to more closely meet all of them. On each iteration, the language will be used in a manner which the final language will be used (e.g., provide inputs to the REVS software). This type of test will identify deficiencies and provide inputs for subsequent iterations. This approach also facilitates incorporation of new SREM concepts and REVS software capabilities within the framework of the RSL. The historical approach to language definition and translator development does not support this approach, and without the advanced techniques described in this section it would not be possible.

6.1 TRANSLATOR DEVELOPMENT TOOLS

Most powerful of the translator development tools available at the ARC are meta-compilers or translator writing systems. These meta-compilers are systems which accept as input the description of source and target languages for the translation and produce as output code for a translator. Code for most of the bookkeeping work necessary in a compiler is added to the translator automatically, reducing the work load on the translator developer. The translator program thus developed is used as a stand-alone program, with no necessity to use the meta-compiler again until the language is changed.

The TRW/TM* Translator [12] writing system is available on the CDC 7600. This TRW proprietary system has been used to develop several translators, including one for a superset of FORTRAN which supports structured programming and one which translates assembly language programs for one machine into ones for a dissimilar computer.

TMG (transmogrifier) is Texas Instruments' meta-compiler, and is implemented on the ASC. This system provides the translation capability used in the Process Design System.

Slightly less powerful than the translator writing systems are macro-processors. The first macro-processor systems were conceived to be processors in which a key word in the input was to be replaced by a predefined string of output. However, the capabilities of macro-processors have evolved over the years to the point where they are able to be used in relatively sophisticated translation tasks. Macro-processors have the advantage that in general they are easier to use than meta-compilers.

The macro-processor ML/1 [14] is available on the 7600; this classic macro-processor was developed at Cambridge University and is considered to have set the standards for subsequent systems. Among the many uses that have been made of ML/I is a natural language input system for a function optimization system called SLANG which was developed at TRW.

Of more recent vintage is STAGE2 [13], a portable macro-processor developed at the University of Colorado explicitly for language development. Because of its portability, STAGE2 is available on both the 7600 and the ASC. It has been used in the implementation of several ALGOL-like languages and also as a bootstrap for itself.

A disadvantage of these translator development tools is that the diagnostic capability of the produced translator is limited. Two approaches are possible which will provide the capabilities described in Section 4. One approach is to manually redevelop the translator at the end of the program after the desired language has been defined and tested using the tools and the iterative procedure described in Section 3. The second approach is to augment the translator produced using the tools with the appropriate diagnostics. These approaches will be evaluated during the REVS preliminary design.

6.2 TECHNIQUES OF IMPLEMENTATION

Several translator implementation tools have been identified for implementing the capabilities required to support SREM. Depending upon the particular REVS software capability (e.g., simulator, emulation, consistency analysis, etc.) different tools or combinations of tools may be employed in conjunction with other existing languages and language processors (e.g., PDL/PDS, PCL/PCP, ECSS, SIMULA, SIMSCRIPT, SALSIM, FORTRAN, SCOPE). Of course, different considerations exist for choosing between concepts for each of the REVS software capabilities, but some of these considerations are illustrated in the following pages where the concepts are explained. For the sake of clarity, only simulation capability is discussed here.

6.2.1 Translation to a Base Higher Order Language (Figure 6-1)

One technique which may be used is to translate the requirements definition to a program in an available higher order language. To ease the translation task, this language would probably be a simulation-oriented language such as SIMSCRIPT or SIMULA since they contain powerful simulation mechanisms. One of the salient features of this technique is that, given the probable disparity in syntax between RSL and the base language, a full-blown translator writing system will be necessary to accomplish the translation. This translation is essentially a mapping of alphas to object descriptions in the base language (probably service routines) and a mapping of the thread descriptions into the sequencing concepts of the language.

The primary advantage of this approach is that it allows great flexibility in the choice of syntax and detailed semantics for the requirements definition language. The implementor has available to him the full power of a simulation language plus the specialized capabilities that can be added by a well-written set of service routines. Other positive factors are the convenience of writing the service routines in a simulation language and the fact that a core of these routines will be all that is necessary, not a different set for each problem.

There are some disadvantages to this technique. Primary among these is the lack of extensibility of RSL. Any additions to the syntax would necessitate regeneration of the translator with the new syntax added to the matacompiler input. Additional concepts (semantics) added would mean the

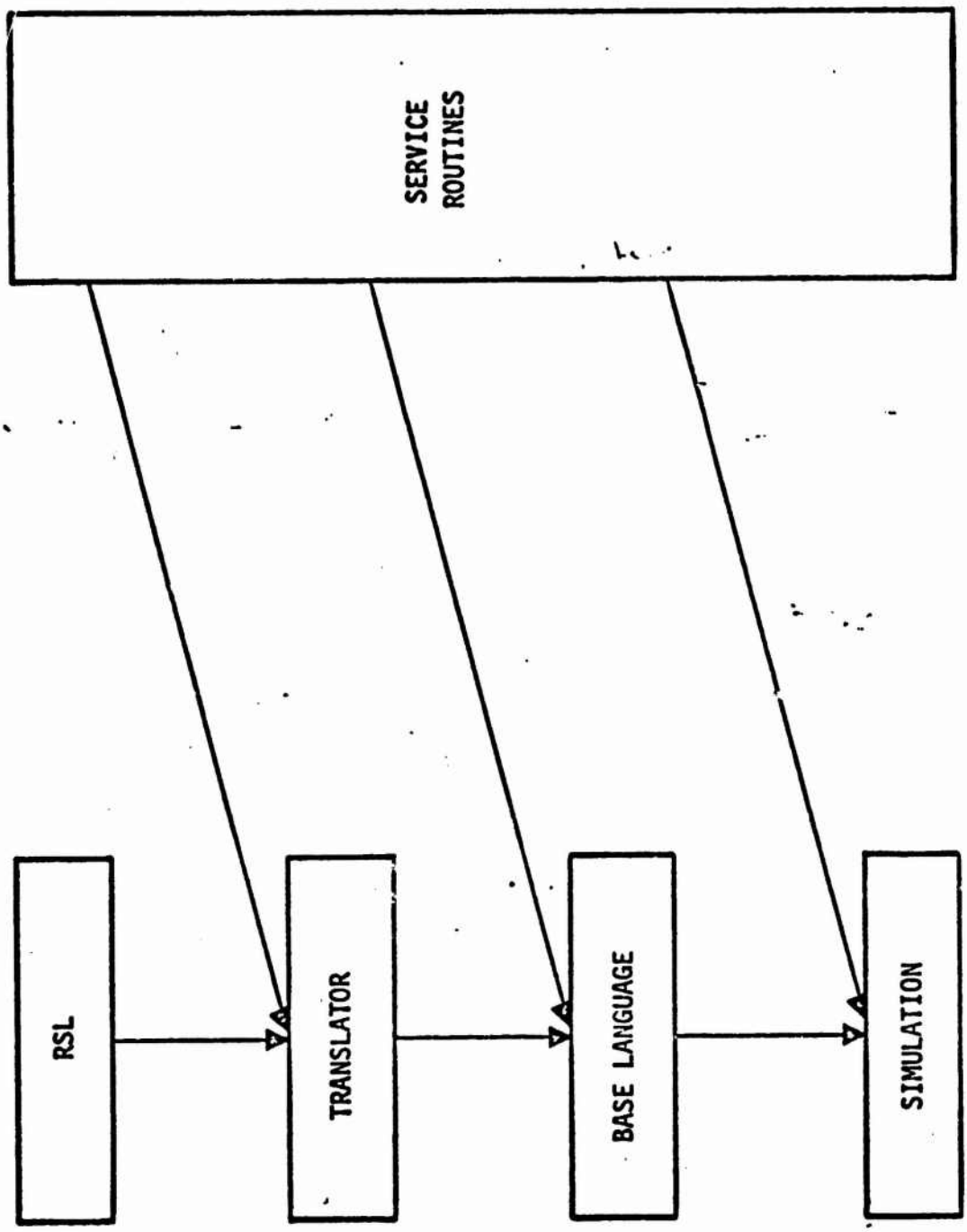


Figure 6-1 Translation to a Base HOL

addition of new service routines. Either of these precludes much tailoring of RSL for each requirements definition task.

6.2.2 Translation to a Series of Modules (Figure 6-2)

Another implementation technique involves translation of the requirements definition into orders which cause a processor to assemble modules from a library. This library would contain general purpose simulation routines and special routines to simulate the characteristics of the object defined. This technique puts much less of a burden on the initial translation step, so a macro processor implementation would be feasible.

The technique of translation to module calls has the advantage of being easier to implement than translation to a base language. Also, if care is taken in defining the module interfaces, existing PDS or PCP libraries, or simulation libraries such as SALSIM, may be used and integrated with routines which already exist to describe the objects of the simulation.

Lack of flexibility in the syntax is characteristic of this approach. The macro processor cannot provide as large an insulator between the source and target syntaxes, and so the source language is constrained to reflect any deficiencies which might exist in the instructions to the simulation builder. Another problem concerns the lack of availability of models for objects in the requirements definition. Each object must have an associated routine or routines in the library. Some amount of time will pass before an adequate base of model routines can be identified and programmed for inclusion in the library. This means that the first few problems for which RSL is used will take additional time for routine development.

A distinct advantage of this approach is the clarity of boundaries between the various modules. Typically, the system can be constructed to facilitate testing individual modules prior to integration into a larger system of interacting modules. In many cases, it is even possible to combine modules that have been programmed in different languages. This is advantageous when the modules are written by individuals with different backgrounds and knowledge of programming languages. (Of course, this can also lead to difficulties in one developer not being able to read the modules written by another developer; standardization is preferable, but not always feasible).

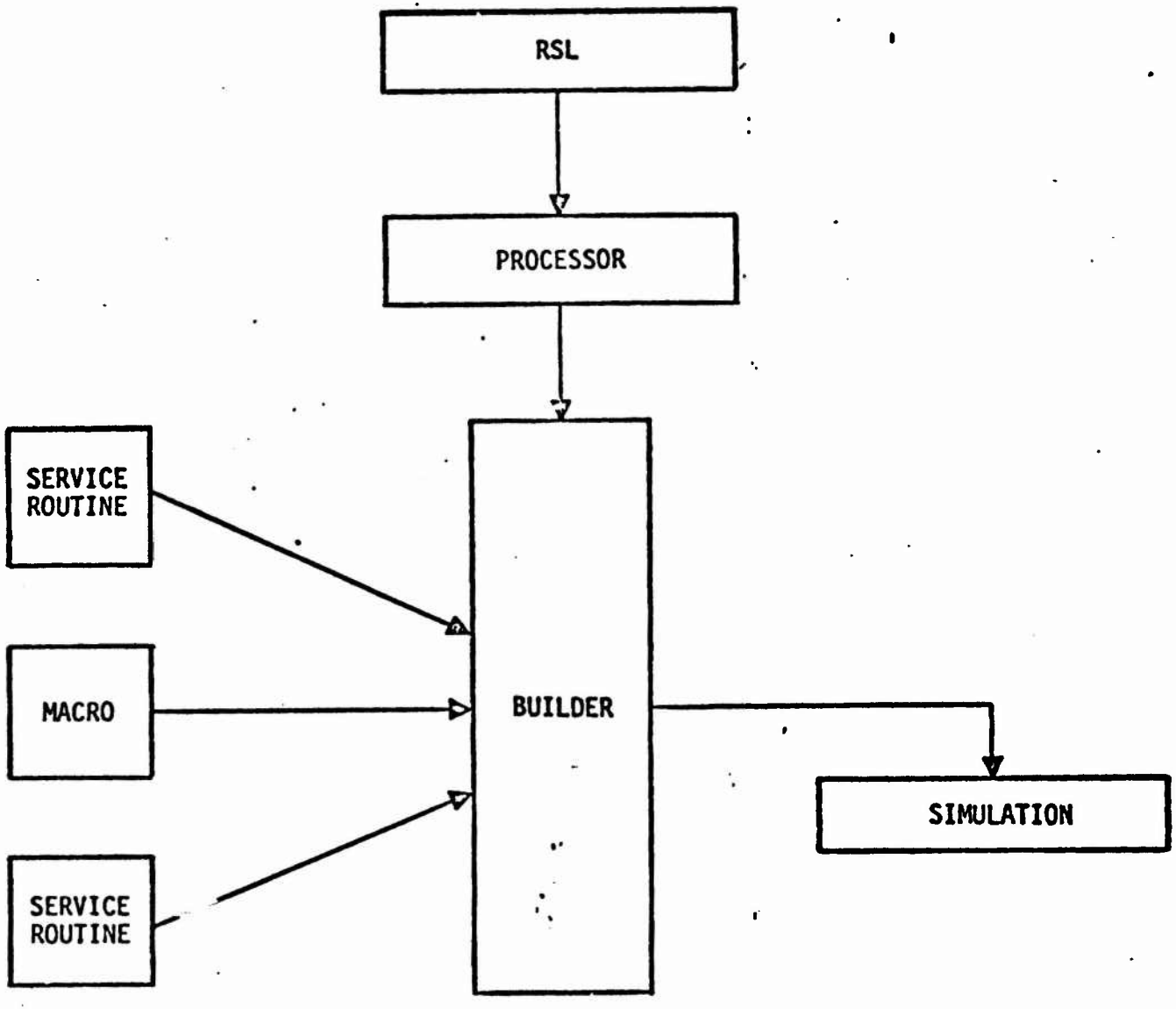


Figure 6-2 Translation to a Series of Modules

6.2.3 Embedding in an Existing Language (Figure 6-3)

The RSL may also be treated as a direct extension to a base language, such as SIMULA. In this case, syntactic and semantic extensions to the base language allow RSL to be run through the base language compiler to produce the simulation. One characteristic of this approach is that the service routines for extension of the semantics handled by the base will be internal to the base language program.

The primary advantage of this technique is that it is the easiest to implement; no additional translators need to be developed. Extensions to RSL provide few additional problems, making this type of implementation the most extensible.

There is one very great disadvantage, however, SIMULA is an extensible language available for the simulation generation processor, but there are very few other extensible languages around. This precludes the use of this technique for the other processors. Also, implementation as a direct extension to a language can provide for little difference between the RSL syntax and the base language syntax. This may mean that RSL syntax is too far from the requirements to be acceptable.

6.2.4 Choice Between Implementation Concepts

The most appealing procedure for choosing between implementation concepts is to (as soon as possible) gather the existing information about strengths and weaknesses, determine feasibility, and then pick one concept over all others. Unfortunately, this procedure is severely inadequate when a number of different capabilities is required and language syntax is not yet known. We plan to refrain from the most appealing procedure and pursue a more effective one. Our plan is to consider the strengths and weaknesses of all the concepts through the point at which preliminary, manual translation has been performed. This plan may require tempering if one or more concepts are shown to be obviously infeasible for a certain capability. If this is so, we will be forced to eliminate that concept from further consideration.

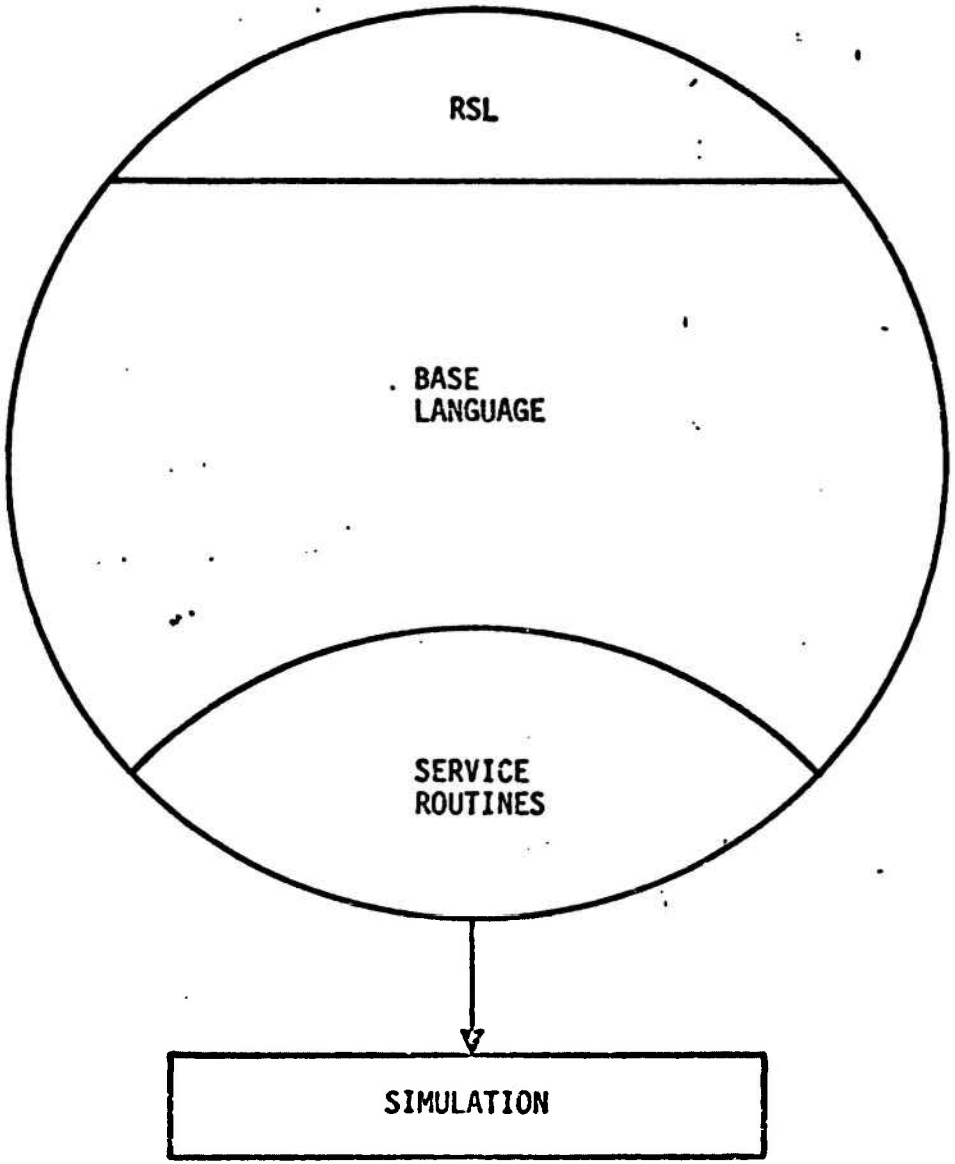


Figure 6-3 Language Extension

This plan breaks with traditional procedures and represents a much more analysis-oriented series of steps than is usually employed by language system designers. However, the difference reflects our orientation toward doing a thorough software system design prior to beginning software implementation. It is an instance of pursuing the same philosophy in our own design and implementation of SREM software capabilities that we advocate should be employed when applying the resultant software to the development of the much larger BMD systems.

7.0 GLOSSARY

Alpha	- A functional unit of Software Requirements. A processing step in the transformation of data from an input port to an output port.
ARC	- Advanced Research Center
ASL	- Algorithm Sequencing Logic
BMD	- Ballistic Missile Defense
BMDATC	- Ballistic Missile Defense Advanced Technology Center
CISS	- Computer-Independent Software Specification
COBOL	- COmmon Business Oriented Language
Emulator	- Non-real-time implementation of tactical software. Incorporates actual transfer functions but operates in a batch environment with a SETS.
FORTRAN	- FORmula TRANslation
JCL	- Job Control Language
JSL	- Job Specification Language
N^2 Chart	- Matrix depicting I/O interfaces (also known as diagonal chart)
PDL	- Process Design Language
PDS	- Process Design System
Port	- External or internal software interface
PPR	- Process Performance Requirements
Processing Step	- An identification of the required processing that must be accomplished in a manner which satisfies stated functional and/or analytical performance requirements (accuracy, precision, format).
REVS	- Requirements Engineering and Validation System
RSL	- Requirements Statement Language
SETS	- System Environment and Threat Simulation
SREM	- Software Requirements Engineering Methodology
SREP	- Software Requirements Engineering Program
Thread	- Sequence of processing steps (alpha's). A port-to-port path in a thread tree diagram.
Thread Tree	- Network of processing steps created by incorporating alpha's with system operating rules and other decision criterion
TSS	- Time Share System

Unclassified
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
TRW Systems - Army Support Facility 7702 Governors Drive West Huntsville, Alabama 35805	UNCLASSIFIED
	2b. GROUP

3. REPORT TITLE

Software Requirements Engineering Methodology Notebook
(Final Report)

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Contract Final Report for period 1 November, 1973 - 31 October, 1974

5. AUTHOR(S) (First name, middle initial, last name)

Mack W. Alford, Earl Benoit, David Bixler
I. Fennell Burns, Jacob C. Richardson, Margaret E. Dyer
Thomas Bell, Ronald J. Smith

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
31 October 1974	260	None

8a. CONTRACT OR GRANT NO	9a. ORIGINATOR'S REPORT NUMBER(S)
DAHC60-71-C-0049 b. PROJECT NO.	22944-6921-020
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d.	CDRL G00C

10. DISTRIBUTION STATEMENT

[redacted]

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Ballistic Missile Defense Advanced Technology Center P.O. Box 1500, Huntsville, Alabama

13. ABSTRACT

The purpose of this notebook is to collect and present in a single document significant research results from the methodology development activities related to the Software Requirements Engineering Program (SREP)--a research program concerned with the development of a systematic approach to the development of complete and validated software requirements specifications. The Software Requirements Engineering Methodology (SREM) is being developed to ensure a well-defined technique for the decomposition of system requirements into structured software requirements, provide requirements development visibility, maintain requirements development independent of machine implementation, allow for easy response to system requirements changes, and provide for testable and easily validated software requirements.

This notebook contains five separate documents (previously delivered to BMDATC for review and approval) which form the basis for a complete SREP methodology description. Included are: 1) a review of current Software Requirements Engineering Technology, 2) a description of anticipated SREM research activities and products, 3) SREM capabilities description, 4) SREM tests and experiments, and 5) requirements on the Software Requirements Language.

DD FORM 1 NOV 65 1473

Security Classification

Unclassified

Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Software Requirements Engineering Program						
Software Requirements Engineering Methodology						
Software Requirements Validation						
Traceability						
Software Methodology						
Requirements Engineering						
Requirements Automation						
Requirements Language						
Threading of Requirements						
Computer Independence						
Process Performance Requirements						
Algorithm Sequencing Logic						
Automated Simulator Generation						

SUPPLEMENTARY

INFORMATION

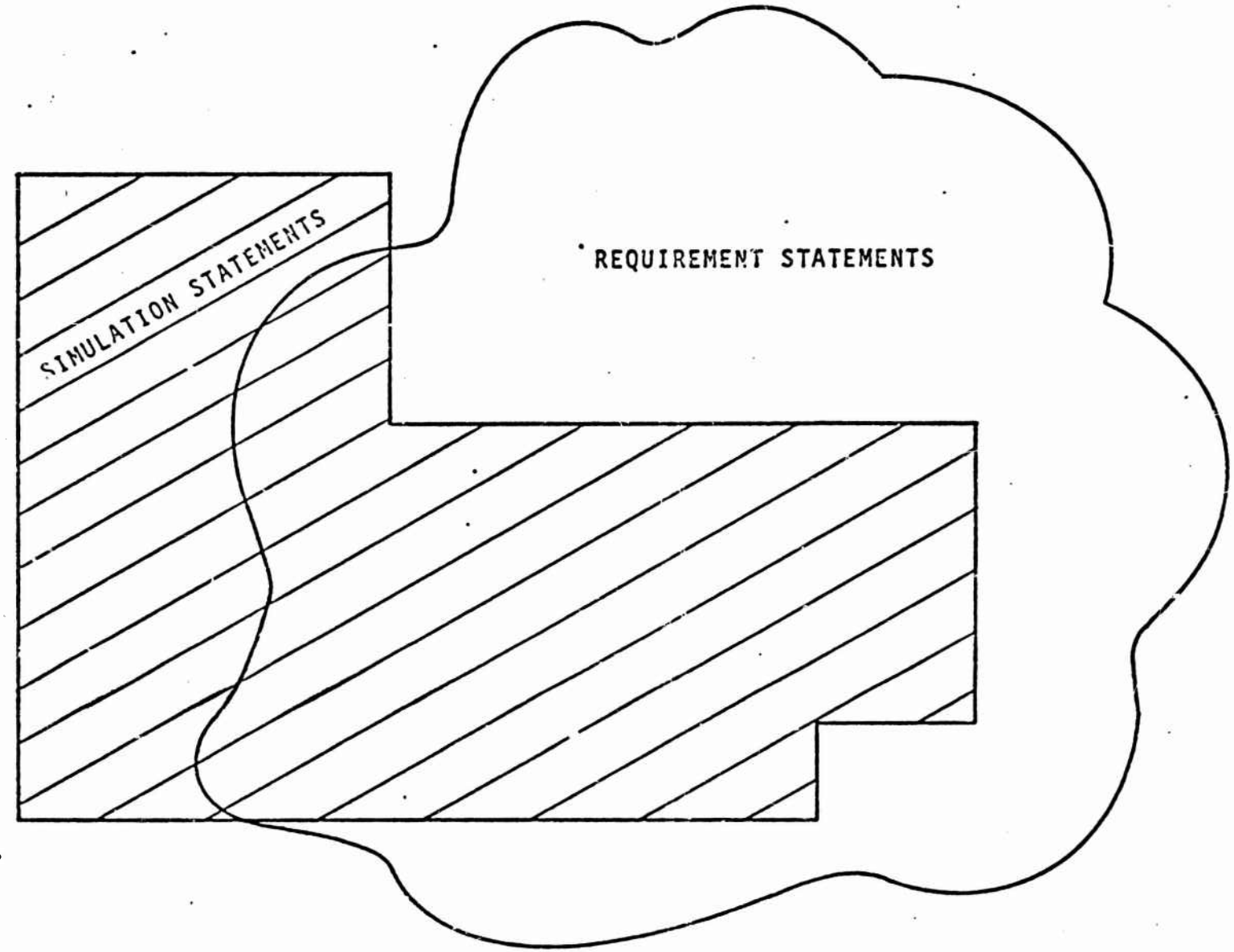


FIGURE 3-1 THE RELATION OF SIMULATION AND REQUIREMENTS